®

d-tree T.M.

DEVELOPMENT TOOLBOX

PROGRAMMER'S GUIDE

d-tree™ Development Toolbox

Programmer's Reference Guide

For Version 3.1

Release E or later

Published by

FairCom
4006 West Broadway
Columbia, MO 65203
(314) 445-6833

First Printing: August, 1988

Table of Contents

THIS PAGE LEFT BLANK INTENTIONALLY

Installation Guide for d-tree V3.1

DEVELOPMENT TOOLBOX

1.1 GENERAL

The purpose of this Installation Guide is to facilitate the creation of your **d-tree** libraries, utilities and example programs. There are three main steps:

- prepare the source code
- compile the source code
- create libraries and executables

Be sure to examine the **Start Up Guide**. In particular, check to see if any source code changes are required. If so, apply these changes to the source files after copying them to your disk but before compiling.

Be sure to examine the READ.ME file in the root directory of Disk 1. Also, notice the following sub-directories on Disk 4 which cover the more popular operating environments.

- \xenix - xenix operation system.
- \unix - unix operation system.
- \msc - Miscrosoft Compiler(DOS)
- \turboc - Turbo C (DOS-Borland)
- \lattice - Lattice C (DOS)

Examine any READ.ME file which may be present in the sub- directory pertaining to your particular environment. Utilization of the make or project file in the appropriate sub-directory will make the installation process much easier. If your environment has not been covered, refer to the **General Installation** section for assistance.

SEE PAGE 1-14 for MOST COMMON ERRORS if you have any problems during installation, compilation or execution.

1.2 Compatibility with Other FairCom Products

ALL ENVIRONMENTS MUST TAKE NOTE

It is necessary to have the **c-tree™ File Handler** installed before you may use the **d-tree Development Toolbox**. We suggest installing other FairCom products, such as the **c-tree™ File Handler** and optionally the **r-tree™ Report Generator** , in sister directories. For example:

c:\ctree
c:\rtree
c:\dtree

Reference the **c-tree** and **r-tree Installation Guide** for detailed installation instructions.
The following c-tree modifications must be applied before installing **d-tree**.

All Systems - Single and Multi User

- CATALOG - NOTFORCE
  Because the **d-tree** CATALOG program does perform rebuilding of files, it is necessary to link this particular program using a buffered version of the **c-tree** library. A buffered (single user) library is accomplished by setting the I/O protocol to **NOTFORCE in CTOPTN.H** before compiling the library modules.
  NOTE: If you require a multi-user (non-buffered) environment, it is recommended that you create a separate multi-user library to be used with your multi-user applications.

- **MAX_KEY_SEG 7**
  The variable **MAX_KEY_SEG** found within **CTOPTN.H** must allow for at least seven key segments per index. Make certain this entry appears as MAX_KEY_SEG 7 (or greater). For more information on key segments see the c-tree documentation.

DOS Systems Only

- **LARGE MEMORY MODEL**- To take full advantage of all the facilities of the "**catalog**" program it is necessary to use the LARGE memory model option when creating the "**catalog**" program.

- Due to the code size of the "catalog" program it may be necessary to reduce the "**sort space**" used by the index rebuild procedures. This may be accomplished by editing the **#define SORT_SPACE** in the **c-tree** file **ctibld.c** to appear as:
  #define SORT_SPACE 16380
  (This is only required for the library used by the catalog. You may start by leaving it as is, but remember this if your rebuild procedure fails due to lack of memory.)

1.3 MS/PC-DOS SYSTEM INSTALLATION

STEP 1: File Transfer
Select the drive (e.g. C) and directory (e.g. \dtree) in which to place the **d-tree** source code. Place Disk 1 in floppy drive A. Type the following command line: *(note the spaces between words. C and \dtree are two separate variables passed to the batch file.)*

a:dtree C \dtree

The batch file dtree.bat will automatically build the necessary directory and sub-directories and copy the source into these directories, prompting you to change disks. After the files have been copied as described above, you will have the following configuration on your selected hard disk directory:

- c:\dtree - contains the system independent code, headers, **d-tree** scripts and utilities
- c:\dtree\xenix - Xenix specific code and headers
- c:\dtree\unix - Unix specific code and headers
- c:\dtree\msc - Microsoft C specific code and headers
- c:\dtree\turboc - TurboC specific code and headers
- c:\dtree\lattice - Lattice C specific code and headers
- c:\dtree\instant - Instant Screen code (direct screen memory access)

NOTE: These files could be created on any drive and with any root directory. For example, if you make \SRC your current working directory on drive E, then type

a:dtree E dtree

E:\SRC\DTREE will be the root directory for your **d-tree** installation.
All of these sub-directories include make files or project files to create the **dtree** utilities and example programs.

NOTE: If you do not wish to install all of these sub-directories, see GENERAL INSTALLATION.

STEP2: Set Environment Variables

Most of the DOS compilers use the INCLUDE and LIB environment variables to specify the directories for system headers and run-time libraries, respectively. Before compiling, be sure to properly set these environment variables based on your compiler's specifications. **d-tree** must also have available the location of the **c-tree** and optionally the **r-tree** header files. This is identified via the **set include** command. This command may be issued at the command line or within the **AUTOEXEC.BAT** file.

C> set INCLUDE = \include;\include\sys;\ctree;\rtree

The global system variable **TERM** must also be set to the appropriate terminal defined in the "**termcap**" file. For most DOS systems, type the command line

C> set TERM=DOS
or for color monitor
C> set TERM=DOSCOLOR

or place the same command in the **AUTOEXEC.BAT** file. *(note: reference the "termcap" section in the reference guide for further discussion of the "termcap" file and utilities.)*

d-tree screen control in the dos environment can be controlled in two ways: either by the use of the ANSI.SYS device driver or by writing directly to video memory.

ANSI.SYS driver- Load this driver in the CONFIG.SYS file as follows:
(load this driver for initial setup)
DEVICE = ANSI.SYS

INSTANT SCREENS - to activate the direct memory video access, set the **#define INSTANT in "dt_defin.h"**. This will activate the direct video logic. Simply comment this #define out to de-activate on non-memory mapped machines.

STEP 3: Testing d-tree settings.

COMPILE AND EXECUTE THE PROGRAM "dt_tests.c" TO VERIFY ALL HEADER SETTINGS ARE CORRECT

STEP 4: Creating Libraries: The next step is to compile the d-tree modules, create a d-tree library, and create some executable d-tree programs. This can be done with either the provided "make" file, or a standard dos batch file. Both are mentioned below. Use one or the other.

Make Files
In DOS systems you must specify a memory model for the compilation. At this time the supplied "make" file should be edited to verify that the proper sub-directories are utilized to reference c-tree and r-tree header files. The provided setup assumes ..\ctree and ..\rtree. Also check the c-tree and r-tree library names used in the provided make. They are defined by CTLIB and RTLIB in the make. To initiate the make in a Microsoft C environment, access the **c:\dtree\msc** directory and then type:

C:\dtree\msc>MAKEDT L

to create a large memory mode set up. This will cause the LDT *(large dtree)* sub-directory to be created and used for all the object files and executables as well as the **dtree** library (called DTREEL.LIB). (That is, c:\dtree\msc\Ldt will contain the results of the make.)
The **d-tree** "catalog" requires that the **large memory model** be used. The "catalog" will only be created if the large model was selected.
Use the large memory model, for initial installation, to compile the "catalog" program. This will allow you to take full advantage of all the features the "catalog" provides as well as allow you to complete the **d-tree TUTORIAL**. After becoming more familiar with the **d-tree** tools you may wish to utilize other memory models. When the large model is selected DTCATLOG.LIB along with DTCATLOG.EXE will be created.

BATCH FILES-
We have provided batch files or in some cases, project files, (Turbo C) to be used instead of make files. In the appropriate directory, review the file DTLALL.BAT or files ending with .prj.

STEP 5 - Complete the Installation: After completing the compilation proceed to **COMPETE THE INSTALLATION** to insure your system is installed properly.

1.4 XENIX SYSTEM INSTALLATION

Step 1 - File Transfer: Select the sub-directory in which to install **d-tree** (e.g.,/usr/dtree). Once you have made this your current working directory, place Disk 5 in your floppy Drive A. Then type the following command line:
doscp a:/xenix/dosmove1 ./
doscp a:/xenix/dosmove2 ./
doscp a:/xenix/dosmove3 ./
doscp a:/xenix/dosmove4 ./
doscp a:/xenix/dosmove5 ./

Then place Disk 1 in your floppy Drive A and type
sh dosmove1

which will copy the files from Disk 1 to the **dtree** directory. Then place Disk 2 in floppy Drive A and type:
sh dosmove2

which will copy the files from Disk 2 to the **dtree** directory. Then place Disk 3 in floppy Drive A and type:
sh dosmove3

which will copy the files from Disk 3 to the **dtree** directory. Then place Disk 4 in floppy Drive A and type:
sh dosmove4

which will copy the files from Disk 4 to the **dtree** directory. Then place Disk 5 in floppy Drive A and type:
sh dosmove5
which will copy the remaining files required for Xenix installation.

STEP 2 - Take out DOS define: Once the files are copied to your machine, edit the file "**dt_defin.h**" as **remove** the **#define** DOS which is located at the top of this file.

STEP 3 - Test d-tree settings:

COMPILE AND EXECUTE THE PROGRAM "dt_tests.c" TO VERIFY ALL HEADER SETTINGS ARE CORRECT

STEP 4 - Creating Libraries: The next step is to compile the d-tree modules, create a d-tree library, and create some executable d-tree programs. This can be done with either the provided make file, or a standard batch file. Both are mentioned below. Use one or the other.

Make File
At this time the supplied "make" file should be edited to verify that the proper sub-directories are utilized to reference c-tree and r-tree header files. The provided setup assumes ..\ctree and ..\rtree. Also verify the library names used in the make represented by the CTLIB and RTLIB symbols.

To make the XENIX applications, execute the shell

sh makedt L

which will create a **d-tree** application library using the **large model** in the file **/usr/lib/Llibdtree.a**, place the **d-tree** objects in **/usr/dtree/Ldt** *(large d-tree)* and place the **d-tree** executable utilities in **/usr/bin**.
The **d-tree** catalog requires that the large memory model be used. The "catalog" will only be created if the large model was selected.
Use the large memory model, for initial installation, to compile the "catalog" program. This will allow you to take full advantage of all the features the "catalog" provides as well as allow you to complete the **d-tree TUTORIAL**. After becoming more familiar with the **d-tree** tools you may wish to utilize other memory models. When the large model is selected **/lib/Llibdtcatlog.a** along with **/usr/bin/dtcatlog** will be created.
The **d-tree** application libraries have been named so that you may use the command line reference
-ldtree

-ldtcatlog

for the library while linking an application program.
In general, the directory structure for **d-tree** is as follows:

/usr/dtree	d-tree source files
/lib	d-tree libraries
/usr/bin	executable modules
/usr/dtree/Ldt	d-tree object modules

Batch Files: There is provided a batch file called "dtreeall" along with a file called "ccc" which may be used to compile the modules without using a make utility. View these files to verify settings (compiler switches, paths, etc.) before using them.

STEP 5: After completing the compilation proceed to **COMPLETE THE INSTALLATION** to insure your system is installed properly.

1.5 UNIX SYSTEM INSTALLATION

STEP 1 - File Transfer: To install **d-tree** on a UNIX system from a DOS diskette, you must have a communications program to connect a DOS and UNIX machine, or a host program *(such as pcdsk)* which can read DOS diskettes. In either case, be sure that the **carriage return characters are stripped-off** the source lines as the code is imported.

The files to copy from the DOS diskettes are as follows:

- all the files on Disks 1,2, 3 and 4
- all the files in the root directory of Disk 5
- all the files in the UNIX sub-directory of Disk 5
- if you do not have r-tree, all the files in the rtreehdr directory on disk 4.

Step 2 - Case Sensitivity: Once the files have been ported to your Unix **d-tree** directory, (ie: /usr/dtree) check to see if the file names are in UPPER or lower case. If they are in UPPER case, use the shellscript

sh ./chgcase -l *

to change the names to lower case *(the "chgcase" shell is on disk 5).*

STEP 3 - Remove DOS define: Once the files are copied to your machine, edit the file "dt_defin.h". Remove the #define DOS which is located at the top of this file.

STEP 4 - Check d-tree settings:

COMPILE AND EXECUTE THE PROGRAM "dt_tests.c" TO VERIFY ALL HEADER SETTINGS ARE CORRECT

STEP 5 - Create Libraries: The next step is to compile the d-tree modules, create a d-tree library, and create some executable d-tree programs. This can be done with either the provided make file, or a standard batch file. Both are mentioned below. Use one or the other.

Make File:
At this time the supplied make file should be edited to verify that the proper sub-directories are utilized to reference c-tree and r-tree header files. The provided setup assumes ..\ctree and ..\rtree. Also verify the c-tree and r-tree library names represented by the CTLIB and RTLIB symbols.

To make the UNIX applications, execute the shell:

$ sh makedt

which will create a **d-tree** application library in the file **/usr/lib/libdtree.a**, the "catalog" library in the file **/usr/lib/libdtcatlog.a**, and place the **d-tree** objects in **/usr/dtree/dt** , with the d-tree executables in **/usr/bin**.

The **d-tree** application libraries have been named so that you may use the
-ldtree
-ldtcatlog
command line reference for the library while linking an application program.

IF YOUR UNIX ENVIRONMENT REQUIRES MEMORY MODEL SPECIFICATIONS, then examine how the Xenix shells incorporate a memory model parameter.

In general, the directory structure for d-tree is as follows:

/usr/dtree	d-tree source files
/lib	d-tree libraries
/usr/bin	executable modules
/usr/dtree/dt	d-tree object modules

<u>Batch Files:</u> There is provided a batch file called "dtreeall" along with a file called "ccc" which may be used to compile the modules without using a make utility. View these files to verify settings (compiler switches, paths, etc.) before using them.

<u>STEP 6 - Complete the Installation:</u> After completing the compilation proceed to **COMPLETE THE INSTALLATION** to insure your system is installed properly.

1.6 GENERAL INSTALLATION

"I Don't Have a MAKE "

For the developer who prefers to compile without the assistance of make files, we offer the following assistance.

STEP 1 - File Transfer: If you do not want all the files from the distribution diskettes to be installed, then you may copy a subset of the files. The root directories of Disk 1, Disk 2, Disk 3 and Disk 4 contain the system independent modules. These modules are necessary for all installations. Then you may choose the appropriate sub-directory on Disk 5 for the remaining files. It you do not have r-tree the header files in the rtreehdr directory on disk 4 must be copied.

STEP 2 - **NON DOS MACHINES ONLY: Once the files are copied to your machine, edit the file "dt_defin.h" as remove the #define DOS which is located at the top of this file.**

Step 3 - Check d-tree settings:

COMPILE AND EXECUTE THE PROGRAM "dt_tests.c" TO VERIFY ALL HEADER SETTINGS ARE CORRECT

Step 4 - d-tree LIBRARY: Create the dtreel.lib *(large model)* by compiling the following modules:

1. DT_IMAGE.c	2. DTPIMAGE.c	3. DT_ALIGN.c
4. DT_FIELD.c	5. DTPFIELD.c	6. DT_MISCI.c
7. DT_CONST.c	8. DTPCONST.c	9. DT_WDODA.c
10. DT_PRMPT.c	11. DTPPRMPT.c	12. DT_UTILY.c
13. DT_SUBFL.c	14. DTPSCANN.c	15. DT_REFMT.c
16. DT_EDITS.c	17. DTPSUBFL.c	18. DT_FREEE.c
19. DT_DFALT.c	20. DTPEDITS.c	21. DT_PSTFX.c
22. DT_CALCS.c	23. DTPDFALT.c	24. DT_TOKEN.c
25. DT_HELPP.c	26. DTPKEYBD.c	27. DT_TSPLT.c
28. DT_MAPIT.c	29. DTPIFILS.c	30. DT_DEBUG.c
31. DT_IFILS.c	32. DTPCALCS.c	33. DT_PARSE.c
34. DT_KEYBD.c	35. DTPHELPP.c	36. DT_COMPL.c
37. DT_DODTS.c	38. DTPMAPIT.c	39. DT_COMPI.c
40. DT_DORTS.c	41. DT_ERROR.c	42. DT_RELAT.c
43. DT_MENUS.c	44. DTPMENUS.c	45. DT_XTRCT.c
46. DT_RTREE.c	47. DTPRTREE.c	48. DT_SPTRS.c
49. DT_CALMP.c	50. DTPTABLE.c	51. DT_INOUT.c
52. DT_EVALU.c	53. DTPHOOKS.c	54. DT_GROUP.c
55. DT_GENRL.c	56. DTPGROUP.c	57. DT_INPUT.c
58. DT_HOOKS.c	59. DT_CTREE.c	60. DT_FRAME.c

61. DT_FUNCT.c	62. DT_SCANN.c	63. DT_EMAND.c
64. DT_EFILL.c	65. DT_EDATE.c	66. DT_ETABL.c
67. DT_EDUPK.c	68. DT_EVALD.c	69. DT_GROUT.c
70. DT_ADDMD.c	71. DT_CHGMD.c	72. DT_MANMD.c
73. DT_MSDOS.c(dos only)		

Note: There are dos batch files in the sub-directory on disk 5 called "dtlall.bat" which may be used to create the library. For unix/xenix there is a file called "dtreeall" which uses the file "ccc" that can be used for compilation.

<u>**Step 5 - Create CATALOG LIBRARY:**</u> Create the CATALOG library by compiling the following modules:

1. DTCATCSP.c	5. DTCATADI.c
2. DTCATADO.c	6. DTCATWAT.c
3. DTCATDIF.c	7. DTCATINC.c
4. DTCATSEL.c	8. DTCATVRT.c

<u>**Step 6 - Create EXECUTABLES:**</u>

Create the executable files by compiling and linking the following source files to the indicated libraries. Use of the large model is required on DOS machines unless otherwise noted. There is a simple batch file using microsoft which can be used as a guide. This batch file ("dtexe.bat") can be found in the msc subdirectory on disk 5.

- **dt_doque.c** - Process Compile Que Program.
  No Libraries Needed. Small Model Ok.
  (NOTE: This executable must be named "dt_doque.exe" EVEN ON NON-DOS Machines. This is because that is the name given in the "dt_compl.h" file which is used to call this program within d-tree . If you do not want to call it this, change the name in "dt_compl.h". See next page.)

- **dt_catalog.c** - Catalog Program.
  Link with dtcatlog , d-tree, and c-tree libraries.

- **dt_catvd.c** - Validation Dictionary Program.
  Link with d-tree and c-tree libraries.

- **dt_score.c** - "run" Program (d-tree super core).
  Link with d-tree, r-tree(optional), and c-tree libraries.

- **dt_bhelp.c** - Build help text index Program.
  Link with d-tree and c-tree libraries.
- **dtcatrpt.c** - Catalog report program.
  Link with d-tree , r-tree and c-tree libraries.

1.7 COMPLETE THE INSTALLATION

<u>Step 1 - Check batch files used for compiling:</u> As you begin to work with **d-tree** you will find that there are options to compile programs during development. There is a group of batch files that are called from within certain programs to accomplish this. The names of these batch files are defined in the d-tree header file called "dt_compl.h". This file is shown below:

```
dt_compl.h
/* batch file compile definitions */
#define   DTCOMPILE "compile.bat"      /* program compile batch file */
#define   DTCATQUE  "dtcompil.que"     /* compile que file */
#define   DTPQNAME  "dt_doque.exe"     /* process compile que program name */
#define   DTCOMP_P  "dtcomp_p.bat"     /* batch file for "run" pgm -c compile
                                          option-parameter files pgm */
#define   DTCOMP_I  "dtcomp_i.bat"     /* batch file for "run" pgm -c compile
                                          option-incremental files pgm */
```

We have placed versions of these batch files in the various subdirectories. If you experience problems when selecting a **compile option** within **d-tree**, we suggest you verify the following files:

compile.bat **dtcomp_i.bat** **dtcomp_p.bat**

Note: We use the .bat extension in all environments. These names can be changed in "dt_compl.h". **UNIX/XENIX**: make these batch files executable as follows:

$ chmod +x compile.bat
$ chmod +x dtcomp_i.bat
$ chmod +x dtcomp_p.bat

SPECIAL NOTE: In these batch files we have made two assumtions which need to be verified.

1) we have set up these batch files assuming you have r-tree. If you do not have r-tree, remove any reference to r-tree in the link statements. By r-tree references we mean any reference to rtintr.obj or to a r-tree library (rtalib or rtsglib)

2) the primary methid of running r-tree scripts is in the interpreted mode. In order to make the batch files simpler, we have assumed that the interpreted version of the report function call (rtintr.obj) has been placed in the r-tree library. Either do the same or modify the batch file to include rtintr.obj on the link line.

STEP 2 - Environment Variables: After completing the compilation as directed for your particular environment make sure that your environment variable TERM has been set as follows:

DOS-
C> set TERM=DOS or **C> set TERM=DOSCOLOR** for color.

XENIX -
$ TERM=ansi;export TERM
UNIX -
$ TERM=vt100;export TERM
(NOTE: substitute "ansi" or "vt100" with the appropriate terminal. Associated entry must be found in "termcap" file. See "termcap" section if more information is necessary).

STEP 2 - Run the dt_catvd Program (MANDATORY): To test the installation run the "dt_catvd" program:

C>dt_catvd

Select the IMPORT DATA option on the menu. This will load the catalog's validation dictionary with data from the text file "dt_catvd.txt". Once import is finished, select option 2, at the prompt enter a zero (0). This will show you a list of valid codes. If this is the case, everything appears to be set up ok. Press "ESC ESC" to return to prompt, ESC ESC again to return to menu. Then press "ESC ESC" to return to operating system. **This import process MUST BE DONE in order for the catlog program to work properly.** The codes that are imported into this file are used to validate proper entry into the catalog.

STEP 3 - Build Help Text Index: The index over the help text file must be built to allow the HELP ability to function in the CATALOG program. **Execute the program "dt_bhelp"** to build this index.

STEP 4 - If Problems: If no errors occur , proceed to the **d-tree** tutorial, else see next page for help.

1.8 MOST COMMON PROBLEMS

The following is a list of possible error messages you may receive if the setup is not correct.

1) **"Error occurred during TERMCAP parse Error = 7201"**
   The "termcap" file is not found in the proper disk and directory.

   "Error occurred during TERMCAP parse Error = 7202"
   The system global variable TERM has not been set or the value it has been set to is not found as a terminal definition in the "termcap" file.

2) **Data Improperly Positioned on the Screen**
   DOS: Make sure ANSI.SYS is in the CONFIG.SYS.

   NON-DOS:If you are operating in a non-DOS environment and the data on the screen is not properly positioned, refer to the **TERMCAP** section within the **REFERENCE MANUAL**.

3) All libraries , d-tree, r-tree (optional), and c-tree, must be compiled in LARGE memory model.

4) Unresolved errors during link:

   a) chkidx unresolved - the c-tree module ctdbug.c must be compiled and placed in the c-tree library. See page 1-2

   b) dtype or report unresolved - the r-tree function report is being called in the program. It needs to find the module rtintr.obj to resolve these definitions. Either place rtintr.obj in your r-tree library or place it one the link line. See special note on page 1-12

5) Error 109 when running a d-tree program. You forgot to set MAX_KEY_SEG in c-tree as mentioned on page 1-2.

6) UNIX Problems - Make sure that all of d-tree has been compiled with the #define DOS taken out of dt_defin.h.

d-tree Tutorial

2.1 The "RUN" Program - Tutorial

Often the best way to learn how to use a tool of any sort is to actually use it. Accordingly, the purpose of this section is to provide a step by step illustration of **d-tree** in action. The objective in doing this is to provide a hands-on example of the power and flexibility of some aspects of **d-tree**. Enough of an example that you, the user, feel comfortable using **d-tree** for simple applications. This should also provide a platform for further investigation of the various aspects of **d-tree** to the extent you feel necessary.

For our purposes of illustrating **d-tree** in action, we will develop a small but relatively complete application. We call this application the *Small Project Accounting System (SPAS)*. Following are some brief specifications of this system. More details will be provided as needed.

The purpose of the *Small Project Accounting System* is to record transactions such as deposits made and checks written along with classification data as to type of income or expense, the project to which the income/expense is related, and customer/vendor involved in the transaction. Once these transactions are collected, they must be able to be reviewed (both on-screen and on paper), and updated. Finally, programs must be able to extract various items of information and create usable reports.

Information for *SPAS* will be organized into the following files:

- **Customer Master File**...customer information
- **Project Master File**........project codes and descriptions
- **Vendor Master File**........vendor information
- **Account Code File**.........ledger account codes and descriptions
- **Transaction File**.............financial transactions
- **Distribution File**.............distribution of trans to accounts

Let's begin our development by creating a file maintenance program for the Customer Master File. Let's assume that this file contains the following fields:

- **Customer ID**
- **Customer Name**
- **Customer Address** (2 lines)
- **Customer City**
- **Customer State**
- **Customer ZIP Code**
- **Customer Phone Number**
- **Customer Type Code**
- **Customer Initial Entry Date**
- **Customer payments year to date**

We will use a utility program provided with **d-tree** called "**run**". This utility assists in quickly generating file maintenance applications. Keep in mind that **d-tree** provides several methods by which to develop programs. We are currently illustrating only one.

To get started, we use our text editor or word processor (be sure that you are able to create pure text or ASCII files, such as non-document mode in WordStar). Create a file named "**customer.dts**". (The file extension, ".dts", stands for *d-tree script.*) We will use the editor to create a sample layout of the screen to be used for data input and display. Only a few special characters must be known for you to do this.

First, data fields will be positioned and sized by the use of underscore ("_") characters. Each underscore is assumed to represent a position for the data field. Additional characters, when used with the underscore, define other aspects of the data field.

For our first example we will need only one additional field definition character - the period or decimal point ("."). When the decimal point is positioned in a field, it indicates that the field is of type real (or floating point). The location of the decimal point identifies the number of decimal digits which are to follow the decimal point in the values contained and displayed in this field. For example, if our field specification is "_____.__", **d-tree** assumes that we wish to have a real value with five integer digits and two decimal digits (for example, 00000.01 to 99999.99).

Another special screen image definition character is the at-sign ("@"). This character is used to identify any of several special system values which can be placed on the screen form.

The current values are:

- **"@DATE"** the current system date
- **"@TIME"** the current system time
- **"@CWD"** the current working directory
- **"@RRNO"** the current relative record number *(requires special handling)*
- **"@SEQ"** the c-tree unique record sequence number *(requires special handling)*

The only other special character for screen layouts is the plus sign ("+"). It is used to draw frames on an image. Placing a "+" where you desire the top left corner of a frame, and then placing an additional "+" at the bottom right will define a frame to **d-tree**. Up to nine frames per image are allowed (note: multiple images per screen are allowed; in addition, this limit of nine can be increased if necessary). Additional frame corners are distinguished by tagging a number to the plus sign ("+2"...."+3"...etc.). Frame titles can also be defined in the following format.

```
          +2This is my Title


                                      +2
```

This will define a second frame and center the title.

Except for these special characters, we may describe the data items, label columns, present instructions, etc. as desired. As you will soon see, the "**run**" program will create a number of default screens. Each of these screens will be given a default *(two line)* title. This *(two line)* title is copied from your screen. In other words, the first two lines you paint in your screen <u>must</u> represent a title. See example. *(Note: This is not a limitation within d-tree as a whole. All screens are not required to have two line titles. This is only used by d-tree's "run" program.)*

OK, let's build our screen image. Using the following field lengths to aid you, simply type the screen as shown on the next page with your favorite editor.

Customer ID:	10	Phone:	14
Name:	40	Type:	3
Address:	40	Last Trans Date:	6
City:	20	Balance:	8,2
State:	2		
ZIP:	10		

```
                              customer.dts
@DATE                 Small Project Accounting System                   @TIME
                        Customer Master Maintenance


          Customer Id:  __________
          Name:         ________________________________________
          Address:      ________________________________________
          City:         ____________________  State: __  Zip: __________

          Phone:  ______________          Customer Type Code: __

          Last Transaction Date: ______   Customer Balance: ______.__
```

Once the screen image is entered and saved we are ready to work with "**run**". Before starting the program, however, be sure that you have an environment variable properly set for your terminal type. For most DOS systems, this can be done from a DOS prompt by typing:

C>set TERM=DOS
(or place this command in autoexec.bat)

You can verify the current setting by typing:

C>set

See your Operating System documentation for further information on environment variables. Be sure that the "**termcap**" file from the **d-tree** distribution disks has been properly installed. *(See "d-tree Installation Procedures".)*

NOTE: On non-DOS systems, you will need to set their TERM environment variable to a corresponding entry in the d-tree TERMCAP file. Example: xenix or unix terminals can be set as follows:

$ TERM=ansi; export TERM

The terminal identifier "ansi" can be replaced with proper terminal (wyse50, etc.)

Assuming **d-tree** has been properly installed, you access "**run**" by entering the command "**run**" followed by the screen image file name ("customer.dts") at a system prompt:

C> run customer.dts

After a few seconds, you should see a menu something like:

```
Sun May 22            Small Project Accounting System            15:38:00
                       Customer Master Maintenance




                           1. Add Records
                           2. Change/View Records
                           3. Delete Record
                           4. Print Records

                  Option:[__]
```

This menu, completely generated by "**run**", identifies the four main activities possible in the file maintenance program which has been generated: **Add, Change/View, Delete, and Print.** As we will see later, these menu item descriptions, as well as much of what was generated with default values, can be easily changed. Note that the screen heading (the first two lines) is taken as the first two lines of our initial screen image.

To check out our file maintenance program, select "**Add Records**", option 1. The next screen that you see should look familiar. It should look something like this:

```
Sun May 22            Small Project Accounting System            15:39:05
                       Customer Master Maintenance



      Customer Id:[__________]
      Name:        ________________________________________
      Address:     ________________________________________
      City:        ____________________ State: __ Zip: __________

      Phone:  ______________        Customer Type Code: ___

      Last Transaction Date: ______ Customer Balance: _________
```

Notice that the field which is expected to be entered is enclosed in square braces ("[ ]"). For the purpose of checking out the File Maintenance Program, enter the following data values:(leave the date and balance fields blank).

Cust Id	Name	Address	City	State	Zip	Phone	Type
11223	Olson, M.	123 Pc Dr.	Los Angeles ,	Ca.	12346	816-665-7865	B
12345	Lemons, Alice	1895 Carter Dr.	Reno,	Nv.	89502	702-322-0238	A
72636	Heady, Janice	212 Code Ave,	Mesa,	Az.	65340	602-765-2243	C

Note that as you enter data, you stay in the **Add** mode. After you "field exit" from the last field, you get a message to "**PRESS RETURN to POST**". This is a "catch" option prior to writing to disk. The **HOME** key can be used to return to the top of the current entry form. It is not necessary to have the user get all the way to the last field in the image in order to post. From any field in the entry form you may use the **POST** key (for DOS press the END key on numeric keypad) to go immediately to the post message.

Once these data records are entered, let's look at the other options from the menu. To do this, exit out of the "ADD" mode by pressing the escape key twice. Back at the Customer Master Maintenance Menu, select option 2 ("**Change/View Records**"). With this option you can select an individual record to view and/or update. Optionally, you may see a list of all (or a specified sub-set) of the records currently in the file. After making the selection for option 2, you should see a screen similar to this:

```
Sun May 22                Small Project Accounting System                  16:34:12
                            Customer Master Maintenance






Enter Desired Record Key:[__________]








                              Press ESC ESC to EXIT
                                FairCom (c) 1988
```

This is called the **prompt** screen. If you wish to retrieve a specific record and you know the key value (for our application, the Customer ID), you simply enter the desired key value. That record will then be displayed.

If you are not sure which record you wish to retrieve, you may enter zero ("0") for the key value. The entry here is used as a "**target**" field to access the file by its key. A value of zero for the target will allow you to browse through a list of all records in the file for the access will start with the first record in the file. (note: a greater than or equal to access is called). Using this method, the next screen will look something like this:

```
Wed May 25              Small Project Accounting System              07:52:51
                          Customer Master Maintenance
Select
   1 11223      Olsen, M.
     123 Pc Dr.                                Los Angeles          CA
     123456     816-665-7865   B   ______ ________
   2 12345      Lemons, Alice
     1895 Carter Dr.                           Reno                 NV
     89502      702-322-0238   A   ______ ________
   3 72636      Heady, Janice
     212 Code Ave                              Mesa                 AZ
     65340      602-765-2243   C   ______ ________
```

If you wanted records whose key begins with "2", enter "2" instead of "0". This method may be used to start listing records somewhere other than the beginning of the file. The **PAGE UP** and **PAGE DOWN** keys allow movement through the file by "pages" (or screens). The **ARROW** keys may also be used to move the displayed list up or down in smaller increments.

At this point, you can select a record to process further by specifying its option number. The selected record will then be displayed in our screen image format. Try both the direct and browse methods of retrieving records.

The Record **DELETE** option allows the user to remove a data record from the file. The record may be selected in the same ways that were used for Change/Review option. After the record is located and displayed, you are asked if this record is to be deleted. Pressing return at this prompt will delete the record, pressing "ESC ESC" will return without deleting.

The **Print** Option requires that you have the **r-tree** library installed. The "**run**" program prepares a default **r-tree** script providing a listing of the contents of the data file. If you have installed **r-tree**, try option 4 which will generate a default **r-tree** report directed to your screen. Just for fun, return to your editor and look at the **r-tree** script created by "**run**". Its name is "**customer.rts**". See all the grunt work we've saved you! This provides an excellent base to start from in building more complex reports.

Review

In this session we created a file maintenance program by first describing the screen image we wish to use for the addition, update, and deletion of records to our file. We used several special characters to describe our screen image, such as:

- _ (underscore) indicates a placeholder for a data field
- "@" (at-sign) indicates the beginning of a system value
- . (period) indicates a decimal point in a floating field
- \+ (plus sign) indicates a corner of a frame

Once we have entered and saved the screen image, we start the "**run**" program with the command line entry:

C> run customer.dts

We are then performing file maintenance over a file containing the data fields described in our screen image. We are able to add, change/view, delete, and print records.

A great amount of work can be accomplished with little effort. This is a sample of the strength of **d-tree**.

Exercise *(NOTE: in order to complete next sessions this must be done)*
1. Create another File Maintenance Program for the Project Master File. Call the **d-tree** script file "**project.dts**". When you have it completed, save it and use "**run**" to see how it works. The Project Master File contains the following fields:

Project Code	(5 positions)
Project Type	(3 positions)
Description	(40 positions)
Contact	(20 positions)
Start Date	(6 positions)
End Date	(6 positions)
YTD Expenses	(8.2 format for dollars and cents)
YTD Income	(8.2 format for dollars and cents)

THIS PAGE WAS LEFT BLANK INTENTIONALLY

THIS PAGE LEFT BLANK INTENTIONALLY

d-tree Tutorial - Session 2

2.2 d-tree scripts - Tutorial

Now that we have seen how easy it is to generate a complete file maintenance program for the Customer Master File (as well as for the Project Master File), let's take a closer look at how "**run**" accomplished this. Bring back the "customer.dts" file in your editor. Right away you will notice that some changes have been made to the file. This was done by "**run**" to provide a complete default script for **d-tree** which implements the file maintenance process.

A new first line has been inserted into the file. It looks something like this:

IMAGE(master) {LSTFLD_ADVANCE}

This line labels the screen image which we created. The default reference name which is given to our screen is "**master**". The information contained in squiggly braces ("{}") specifies options. For a thorough discussion of these options, see the Ability Reference Section.

Next is our screen image. This has not been changed. However, following it are more new script entries. We will quickly look at them but for a detailed description, see the **d-tree** Ability Reference Section. The next set of script lines looks like this:

IFILS FILE_NAME customer
KEY_NAME customeridx
KEY_FIELDS F0001

These items describe the incremental file structures ("IFILS"). These names are the symbolic references which will be used within the script for the data file and associated indexes. The data file symbolic reference is "**customer**". The actual file name is found on disk as "**customer .dat**". Similarly, the index file symbolic reference is "**customeridx**" while the disk file name is "**customer.idx**". As a default, "**run**" assumes that the first field specified is to be a "key" field. It specifies this with the KEY_FIELDS keyword. The first field is given the default name of "F0001" (see next page).

The next section of script looks like:

```
                         customer.dts
FIELD(master)
/* Symbol Name Input Atribute Output Atribute  Input Order  Special */
   F0001          NONE          NONE               1 /* Customer Id: */
   F0002          NONE          NONE               2 /* Name: */
   F0003          NONE          NONE               3 /* Address: */
   F0004          NONE          NONE               4 /* City: */
   F0005          NONE          NONE               5 /* State: */
   F0006          NONE          NONE               6 /* Zip: */
   F0007          NONE          NONE               7 /* Phone: */
   F0008          NONE          NONE               8 /* Customer Type Code: */
   F0009          NONE          NONE               9 /* Last Trans Date:*/
   F0010          NONE          NONE              10 /* Customer Balance: */
```

The "**FIELD**" keyword identifies this section which specifies: data fields used on the screen, special attributes for input or output of the field, the input sequence, and a comment which contains the label which preceded the data field on the screen.

Since multiple screens may be needed within the same **d-tree** script, each **FIELD** section is tied to a specific **IMAGE** section. This is done by the parameter which follows the "**FIELD**" keyword (in our case "**master**").

Each data field in our screen image is given a **symbolic name**. The default names begin with the letter "F" followed by a four digit sequence number. These names can be changed, as long as you're careful. Be sure that changes are made consistently throughout the script. One special task is also required to change these names. Changing the names in this FIELD section will define the symbolic names in the DODA (described later). In order for the field names defined in the IFILS structure to be valid the IFILS definition block must be moved below the FIELDS defininition block *(note: this only pertains to the "run" program when field names are changed. This is not a general rule in d-tree.)*

The next column contains **Input Attributes**. The default created for us by "**run**" is "**NONE**". These values can specify various character level edits, attributes, and formats for handling the input values.

For example, if we wish to use alphas in the "Customer ID" field, it would be useful if the system would automatically shift any lower-case alphabetic characters to upper-case. This can be accomplished by replacing the "**NONE**" keyword under "Input Attributes" for "F0001" (Customer ID) with "**ALLCAPS**". The same could be done with the "State" field (F0006). Another useful function would be to capitalize only the first letters of each word in the "Name" field. This is specified by placing the "**ALLWORDCAPS**" keyword in the Input Attribute for field F0002. More specific character level editing can be done with field **MASKS**. These are defined as an additional attribute.

Try making these changes to your script. For more information on other **Input Attributes**, **Output Attributes**, and **MASKS**, please see the **FIELD** keyword definition in the Abilities Reference Section.

Input Order defines the direction the cursor will travel when maintaining this image. The cursor simply travels from field to field in this order (ie: 1 is the first field, 2 is the second, etc.) The default order is top down, left to right. Try changing the order and notice the cursor flow when the program is rerun. You will be asked to rerun the program after a few more steps.

The next section of the **d-tree** script looks like this:

```
customer.dts
EDITS(master)

Must Enter Field F0001 MANDATORY
This record Already Exists F0001 DUPKEY customer_idx
```

This section provides "full field" editing of the values entered, as opposed to character level editing provided by the FIELD keyword. Again, since multiple sets of edits may be included in a single **d-tree** script, we specify which **IMAGE** these **EDITS** pertain to by specifying the reference "**master**".

Two edits are provided with the default script which "**run**" generated. The first specifies that field F0001 is "**MANDATORY**". While in this field, if the user attempts to field exit without having entered information, the message "**Must Enter Field**" will be displayed in the error message area.

The second edit specifies a check for duplicates ("**DUPKEY**") on field F0001. This is performed by checking the key index "customer_idx" for a match on the value entered for this field by the user. If a match is found, the key has already been assigned to an existing record and therefore cannot be added with this record. If a duplicate condition is detected, the message "**This record Already Exists**" appears in the error message area.

For our application, let's add an edit to mandatory fill the state field. (F0005). This is done by adding a new line to the EDITS section. The general format of this line is:

———Error Message———	FIELD ID	EDIT TYPE
State Must Have Two Characters	**F0005**	**MAND_FILL**

Enter this line into your **d-tree** script right after the current DUPKEY edit.

The remaining script sections contain additional specifications about how to handle:

- "browse" mode for retrieving records for update or deletion.
- menu for selecting mode in which the program is to operate.
- various other Abilities.

We shall discuss these in detail later in this session. Also, you may find each keyword in the Abilities Section of the **d-tree** Reference Manual.

That's enough changes for now, let's rerun our script and check the effect of the changes we've made. Exit your editor after saving the file. Then invoke the "**run**" program again as follows:

C> run customer.dts

When you get to the main menu, select the "**Add Record**" option. Try entering lower-case alpha characters as part of the "Customer ID" and "State" fields. Notice that the system automatically converts any input to upper case. Next enter only one character in the state field. An error message should appear detecting this error. Notice cursor flow if you changed the order.

In general, modifying **d-tree** scripts is that easy! With only a basic understanding of the details of the script definition, you are able to customize a file maintenance program to your needs. Let's take a quick look at the remainder of the default script generated by "**run**". Use "esc esc " to exit the File Maintenance Program then reload the script file into your editor.

The next section following our "EDITS" section is as follows:

```
                                customer.dts
IMAGE(heading) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                       Small Project Accounting System              @TIME
                              Customer Master Maintenance
Select









                            Enter Desired Option: __
                               Press ESC ESC to EXIT
                                 FairCom (c) 1988
```

This screen **IMAGE** forms part of the screen which is used to show multiple records in browse or "**SCAN**" mode. It provides the heading portion of the screen as well as the input area for the selected option.

Note that the **IMAGE** is given a reference name ("**heading**") different from that of our first image ("**master**"). This is necessary in order to uniquely identify each image within the script. As you might suspect, the "**heading**" image has an associated "**FIELD**" specification. This is the next section in the script:

```
customer.dts
FIELD(heading)
/* Symbol Name   Input Atribute  Output Atribute  Input Order  Special */
    option          NONE               NONE             1
```

This section specifies only one field, named "option". The field "option" is a global variable used by **d-tree**. It is used to enter the selection number from the browse screen.

The next section specifies another image which shows the format of the records which are shown within the "**heading**" image. This image looks like:

```
customer.dts
IMAGE(rollpart) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP}

  __ _________ ________________________________________
     ____________________________________ ____________________ __
     _________ ______________ __ ______ ________
```

Notice that the underscores specify the size and location of all of the fields in a record, just as with our screen image. Since "**run**" was not able to fit fields of the record on one line, it broke the record into four lines. We can, of course, edit this image to make it fit our needs. Care must be taken to reflect any changes to the **IMAGE** in the upcoming **FIELD** specifications. Let's look at that first.

```
customer.dts
FIELD(rollpart)
/* Symbol Name Input Attribute  Output Attribute Input Order  I/O Mask */
    counter        NONE              NONE           1
    F0001          NONE              NONE           2 /* Customer Id: */
    F0002          NONE              NONE           3 /* Name: */
    F0003          NONE              NONE           4 /* Address: */
    F0004          NONE              NONE           5 /* City: */
    F0005          NONE              NONE           6 /* State: */
    F0006          NONE              NONE           7 /* Zip: */
    F0007          NONE              NONE           8 /* Phone: */
    F0008          NONE              NONE           9 /* Customer Type Code:*/
    F0009          NONE              NONE          10 /* Last Transn Date:*/
    F0010          NONE              NONE          11 /* Customer Balance:*/
```

First note that this **FIELD** specification is tied to the **IMAGE** we just studied, by the reference name "**rollpart**". This specification shows eleven fields, one for each of the fields of our data record plus one for the "counter". The field "counter" is another global work field provided by **d-tree**.

Now let's modify the "**rollpart**" IMAGE and FIELD specifications to show only the counter, the customer ID, name, and type. In other words, let's drop all but what is essential to uniquely identify an individual customer so that we can fit the remaining fields on one line.

We first use our editor to delete the underscores for the fields we wish to eliminate from our scroll-part. Then we move the underscores for the customer type field up to the same line. Next we must remember to delete the corresponding line from the field specification. We should also renumber the Input Order entries. Once these changes have been made these sections should look like:

```
                           customer.dts
IMAGE(rollpart) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP}



  __ _________ ______________________________ ___

FIELD(rollpart)
/* Symbol Name   Input Attribute  Output Attribute  Input Order   I/O Mask */
    counter           NONE             NONE            1
    F0001             NONE             NONE            2 /* Customer Id: */
    F0002             NONE             NONE            3 /* Name: */
    F0008             NONE             NONE            4 /* Customer Type */
```

Let's continue our look at the remainder of the script before running it. The next section is shown here:

```
                           customer.dts
IMAGE(prompt) {INPUT_ADVANCE=1}
@DATE                  Small Project Accounting System              @TIME
                        Customer Master Maintenance




Enter Desired Record Key: ________




                          Press ESC ESC to EXIT
                            FairCom (c) 1988

FIELD(prompt)
/* Symbol Name   Input Atribute  Output Atribute  Input Order  Special     */
   F0001             NONE            NONE              1
```

This section provides what is called the "**prompt**" for retrieving records. You probably remember the format from our use of the Change/View Records and the Delete Record options from the main menu. Notice again that the **IMAGE** and **FIELD** sections are tied together with the reference name "**prompt**".

The "**run**" program defaults to the prompt phrase "Enter Desired Record Key:". As with much of **d-tree**, this may be changed to fit your specific needs. Why don't we change this default to read "Enter Desired Customer ID:".

The next section should also look familiar:

```
customer.dts
IMAGE(menu) {LSTFLD_ADVANCE}
@DATE                       Small Project Accounting System                @TIME
                              Customer Master Maintenance



                                   1. Add Records
                                   2. Change/View Records
                                   3. Delete Record
                                   4. Print Records

                          Option: __

                                Press ESC ESC to EXIT
                                  FairCom (c) 1988

FIELD(menu)
/* Symbol Name   Input Atribute  Output Atribute  Input Order  Special */
    option          NONE              NONE              1
```

This section obviously describes the menu screen, complete with the "option" which is selected. The default wording used for the menu selections may also be modified. Change the option descriptions to read:

1. Add New Customers
2. View/Update Customers
3. Remove Customers
4. List Customers

The final sections are a little less obvious. First:

```
customer.dts
SCAN(master) {IMAGE_OUT=heading} {IMAGE_ROL=rollpart} {IMAGE_INP=heading}
```

This script entry controls the browse mode for retrieving records. The "**SCAN**" keyword sets this up. The "**SCAN**" reference name, defaulted to "**master**", is used to identify this "**SCAN**" to the "**PROMPT**" section below.

The "**SCAN**" syntax specifies both a fixed screen portion ("**IMAGE_OUT=heading**") and a scrolling portion ("**IMAGE_ROL=rollpart**"). It also describes which **IMAGE** is to be used for input ("**IMAGE_INP=heading**"). In each case, the symbolic reference is used to tie the various components together.

The last section in the default script controls the retrieval process. This "**PROMPT**" is identified as "**master**". The IMAGE to be used for prompting is specified with the "**USES_IMAGE**" keyword. In our case, the IMAGE labeled "**prompt**" will be used to request a value for the retrieval key. The key ("customer_idx") to be searched (if "fields for target" match) is specified followed by the name of the scan routine ("master") to be used if an exact match on the key is not found. Finally, the field(s) which relates to the search is specified as "F0001". This is the field (must be defined in the "USES_IMAGE" image) that is used to determine if this is the proper key and scan to use. As you will see later most prompts have more that one key and scan alternative. If values have been entered into the "fields for target" field(s) (in this case just one-F0001) then this index and scan will be used. The PROMPT keyword has some powerful methods for building targets to access data. See the Abilities Reference Section for complete details.

```
customer.dts
PROMPT(master)
 USES_IMAGE(prompt)
/* key symbol name  scann name   fields for target      prefix */
   customer_idx                master                  F0001
```

We made some further changes since we last used "**run**" with our modified script, so save the changes, exit your editor and rerun the script to check out the changes made to the retrieval screens.

Review

In this session, we have looked at some details of how "**run**" uses the **d-tree** script to quickly implement a file maintenance function. We saw that by making easy modifications to the script we are able to customize the "**run**" program to meet our needs.

Exercise (this exercise is optional to complete tutorial)

Modify the **d-tree** script which you generated for Project Master File Maintenance. Include the following changes:

a) for the Project Code field, force alpha characters to upper case. (ALLCAPS)

b) for the Contact field, add a new input attribute which specifies the automatic capitalization of the first letters of each word. This is done by replacing the NONE input attribute with ALLWORDCAPS.

c) for the YTD Expenses and YTD Income fields, specify an input attribute of NUMERIC.

d) for the Project Type field, add a "TABLE" edit.

The "**TABLE**" edit format is:

Error Message   FIELD SYMBOL TABLE value1 value2 ...

Example:

Project Type must be 'BIL' or 'NON' F0002 TABLE BIL NON

e) for the Start Date and End Date fields, add a MMDDYY date edit to the edits section.

Example:

Invalid Date F0005 DATE_MMDDYY

Invalid Date F0006 DATE_MMDDYY

THIS PAGE LEFT BLANK INTENTIONALLY

d-tree Tutorial - Session 3

2.3 The Catalog - Tutorial

The "**run**" program was intended as an example mainline, providing a fast method to complete a simple application. It is useful for prototyping, development and maintenance (when you need some test data in a file fast, or you need fast maintenance to fix a flag in a file, etc). It's primary purpose is to illustate the dynamics that are possible using the tools. Later sections will go into the tools that were used in "**run**". That's where the **true power of d-tree** lies.

d-tree provides several ways to perform file maintenance. In this session, we are going to work with another application written with the tools called the **catalog**. The **d-tree** catalog provides a consistent way of maintaining detailed information about our applications. There are several sections to the catalog: the data dictionary, the program dictionary, and the relate dictionary are the most evident. (For details on these and other features of the catalog, see the Catalog Section of the **d-tree** Reference Manual.)

One of the useful things that the **d-tree** catalog does is to automatically create program specifications not only for simple single-file maintenance programs, but also involving multiple- file update programs. This saves the programmer a lot of detail work. In our exercises we will create programs using several files. Some will be used to validate fields and others as subfiles or update files. In order for the catalog program to do this, all files must first be defined in the data dictionary.

The catalog provides the capability to import file definitions based on **d-tree** scripts which may have been created by the "**run**" program. Let's see how this is done as well as look at some of the features of the catalog itself.

Before starting the catalog, be sure that your TERM environment variable is properly set as described is Session 1. Also be sure you have imported the validation data into the dt_catvd files as described in the installation guide.

To activate the catalog from the system prompt, enter:

C> dtcatlog

It will take a little time for the catalog to finish loading. The reason for this is that the catalog will parse in its ALIBITY definitions from a **d-tree** script.

After the first run, the catalog will load much faster since it has stored the results of the parsing into the dictionary itself. You might find it interesting to analyze the catalog script which is in the file "DTCATLOG.DTS".

The catalog will display the main menu screen as shown below:

```
Sat Apr 28                     FairCom                        07:37:28
                            System Catalog


 Master Menu
 Data Dictionary
 Program Dictionary
 Relate Dictionary
 Execute Compile Que
 Clear Compile Que
 Catalog Reports
 Return to OS




                          FairCom (c) 1988
```

For the demonstration of importing our "**run**"-generated scripts, select the "Program Dictionary" selection from the main menu. At this point, a secondary menu will "pop-up" which looks like this:

```
Sat Apr 28                     FairCom                        07:37:28
                            System Catalog


 Master Menu
 Data Dictionary
 Program Dictionary    <-----
 Relate Dictionary              Add
 Execute Compile Que            View/Modify
 Clear Compile Que              Delete
 Catalog Reports                Import Pgm
 Return to OS




                          FairCom (c) 1988
```

For now, select the "Import Pgm" option from the secondary menu. This will prompt you for the **d-tree** script name. First, import the "customer" script (do not enter extension..the default of .dts will be used). When everything has been added to the dictionary, the catalog program will parse in the script created by the "**run**" program and begin executing your program from within the catalog environment . An interesting thing has just happened which illustrates one of the

most powerful aspects of the **d-tree** tools. Applying the tools in the proper manner allows the user to create data independent mainline programs where the data file as well as the ABILITY definitions can be swapped in and out of memory. We have just freed the catalog definition and loaded in the customer definition. The program now executes exactly like the "**run**" program, but you are actually in the catalog mainline. Exit the customer maintenance with "esc esc". The customer definition will be freed, and the catalog definition will be reloaded from the dictionary, returning you to the main menu of the catalog.

Now let's look into the "Data Dictionary" to see what was imported. Select "Data Dictionary" from the main menu and "View/Modify" from the secondary menu. You will next be asked for a file name to work with. You may specify the exact file name or, if you aren't sure, enter "0". (Remember the browse mode in our SPA maintenance programs?) This will provide you with a list of files for which data dictionary entries are currently maintained. Simply select the number in front of the "customer" file name.

The next screen you see should look like this:

```
Sat Apr 28                                   FairCom                        07:42:50
File Name:[customer    ]       File Description: customer
Version Number: 1              System Name:      IMPORTED
File Type:                Extension: 4096 Mode:  1 Rcd Len:   162 Indexes:    1
 Field                         Field                                  First Vlen
 Name          Type Len Dec    Description                              Field
 F0001           A   11  _     Customer Id:                              ___
 F0002           A   41  _     Name:                                     ___
 F0003           A   41  _     Address:                                  ___
 F0004           A   21  _     City:                                     ___
 F0005           A    3  _     State:                                    ___
 F0006           A   11  _     Zip:                                      ___
 F0007           A   15  _     Phone:                                    ___
 F0008           A    4  _     Customer Type Code:                       ___
 F0009           A    7  _     Last Transaction Date:                    ___
 F0010           DF   8  _     Customer Balance:                         ___
 ____________  __  ___  _     ______________________________________     ___
 ____________  __  ___  _     ______________________________________     ___
 ____________  __  ___  _     ______________________________________     ___
 ____________  __  ___  _     ______________________________________     ___
 ____________  __  ___  _     ______________________________________     ___
 ____________  __  ___  _     ______________________________________     ___
                                 Press ESC ESC to EXIT
```

Notice that the catalog has made entries into the data dictionary. Entries have also been made to the program dictionary. In addition, **d-tree** records the relations between data and program in the relation dictionary. The catalog has picked up a complete description of our application and its data. With the reports available from the catalog, this provides excellent documentation and control over data specifications.

Once both program and data dictionary entries have been made in the catalog, **d-tree** can help maintain and control changes. One way **d-tree** can do this is by identifying where a specific field (or file or index, etc.) is referenced. This allows for easier estimation of the magnitude of a change as well as expediting the actual implementation of the change. For more details, see the Reference Manual section on the catalog and its reporting functions.

But that's not all! Let's assume that we need to change a field size in an existing data file. Before **d-tree**, this would involve many hours of recoding, writing special conversion programs, testing everything, then finally switching over. With **d-tree** this type of change is handled with minimal effort.

The catalog provides more direct assistance for these maintenance situations. Since we are in the Data Dictionary with our "customer" file in browse/modify mode, let's change the size of the Customer Type field. It is currently 3 characters long. Let's assume that we need to make it 6 characters long. First though, notice that the lengths indicated on the data dictionary screen are one greater than what we specified for "**run**". This is due to the need for string lengths to include space for a null byte to terminate the string in the " C" language. We must remember to **add this extra character** to the length of any strings when entering field definitions from the data dictionary.

Simply use the arrow keys to position yourself at the length field for F0008. Then change the length entry from 4 to 7. Now use the POST key (for IBM-PC use the END key - or see termcap for proper key) to accept the change.

When you are in modify mode in the data dictionary and make any change to the file definition, the catalog will know it. If any change has been made, the catalog asks if this is a new version (with its own version number) or a replacement for the same version number. If you specify (N)ew, you must have already changed the version number. If you have not, the catalog will display the following error message:

New Version of File Definition Must Have New Name Or Version Number

and return you to the main file description screen. If, in fact, you want a new version, change the version number (for example, to 1.1) and use the POST key again.

For our example, specify (R)eplace. With this option you will be given a chance to reformat the file from the old format to the new format. The catalog also gives you the option of creating a stand-alone executable program that can be used for converting files from the old format to the new format at any time. This provides a package that you can supply to your customers to automatically update their files. These options are presented with the following screen:

```
Sat Apr 28                            FairCom                             08:15:02
                               File Reformatting Utility

                    The Old File Definition is Will Be REPLACED.



                The file format facility requires both the old and the new file
                layouts. If you want to take advantage of the file reformat
                facility one of the following actions must be selected:

               [_]Reformat the following file in place.

                _ Create a stand alone executable to be used for
                  reformatting at a later time.


                         Place a 'Y' in desired option(s) then
                              Hit POST key to continue.

                                FairCom (c) 1988
                           Press ESC ESC to CANCEL REPLACE
```

Since the old file format is **lost forever**, if you do not choose one of these options, automatic reformatting will be difficult. In fact, you will probably need to rekey the old file format into the catalog. *(NOTE: both options may be selected)*

Because we are in an exercise, let it reformat the file in place by placing a 'Y' in the first option. You will next see the following screen which shows the mapping of the old format into the new format:

```
Sat Apr 28                            FairCom                             08:15:19
File Name: customer            File Description: customer
Version Number: 1              System Name:      IMPORTED
File Type:               Extension:  4096 Mode:  1 Rcd Len:   164 Indexes:    1


  ******* Old File Layout *******     |*|   ***** New File Layout *****
 Map  Name........ Typ Len Off Var    |*|  Map  Name........ Typ Len Off Var
   1. F0001         12  11   0   0    |*|    1. F0001         12  11   0   0
   2. F0002         12  41  11   0    |*|    2. F0002         12  41  11   0
   3. F0003         12  41  52   0    |*|    3. F0003         12  41  52   0
   4. F0004         12  21  93   0    |*|    4. F0004         12  21  93   0
   5. F0005         12   3 114   0    |*|    5. F0005         12   3 114   0
   6. F0006         12  11 117   0    |*|    6. F0006         12  11 117   0
   7. F0007         12  15 128   0    |*|    7. F0007         12  15 128   0
   8. F0008         12   4 143   0    |*|    8. F0008         12   6 143   0
   9. F0009         12   7 147   0    |*|    9. F0009         12   7 149   0
  10. F0010          8   8 154   0    |*|   10. F0010          8   8 156   0




                           Press RETURN to START RE-FORMAT
```

Pressing 'RETURN' will execute an in-place file reformat. You will see the messages that the reformat as well as the index rebuild is in process. Those of you who have used c-tree's rebuild facility will recognize these messages. The file reformat has NULL'd out the header portion of the c-tree file. The rebuild will reconstruct the entire header. Because the header has been NULL'd out the following message may appear. Simply answer yes (Y) to the prompt for the rebuild to continue.(**note: you must respond with a yes. Failure may result in a divide by zero run time error.**

```
Examining data file customer.dat.

WARNING: data record length discrepancy.
Parameter file data record length = 161
        Header data record length = 0

Use parameter file value? (y or n)>> y

WARNING: file extension size discrepancy.
Parameter file file extension size = 4096
        Header file extension size = 0

Use parameter file value? (y or n)>> y
```

The catalog rebuilds files by calling c-tree's RBLIFIL function. It is this function that is displaying these messages. These messages can be suppressed by editing the c-tree file ctrbl2.c and setting the proper #define's. The following is a snap shot of this c-tree source file.

```
ctrbl2.c
/*
 * To disable interactive rebuild prompts and/or to suppress normal
 * rebuild output, "comment out" one or both of the following
 * defines:
 */

#define RB_INTERACTIVE
#define RB_OUTPUT
```

Note: Something to keep in mind later as you work with d-tree. You will be creating many d-tree scripts for different programs as you develop with d-tree. If you modify to a file structure, such as changing a field length, you will need to go back into the scripts which contain a reference to this field, and make the necessary changes (i.e. change input length of the field on the screen).

Let's create another file specification, but this time we will use "**dtcatlog**" instead of "**run**". We need a file maintenance program for the Vendor Master File.

This file should contain:

• Vendor Code	10 positions
• Vendor Balance	8.2 format for money
• Vendor Name	40 positions (first vlen field)
• Vendor Address (2 lines)	40 positions each
• Vendor City	20 positions
• Vendor State	2 position
• Vendor ZIP Code	10 positions
• Vendor Type Code	4 positions

Enter the catalog program and select "**Data Dictionary**" from the main menu. Then choose "**Add**" mode. The following screen will then appear:

```
Sat Apr 28                           FairCom                              10:52:17
File Name:[____________]    File Description: ________________________
Version Number: ______      System Name:      ____________________
File Type: ____________ Extension: _____ Mode: __ Rcd Len: _____ Indexes: ___
 Field                      Field                                    First Vlen
 Name         Type Len Dec  Description                                 Field
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
 ____________ __   ___  _   ________________________________________     __
                              Press ESC ESC to EXIT
```

Assign the file name of "vendor". Enter an appropriate Description, such as "Vendor Master File Maintenance". The version should be "1" or "1.0". The system name is "Small Project Acctng". File type is "MASTER". The extension specifies the file size extension of the data file (c-tree). For now, hit the default key (TAB key).

The **mode** specifies the c-tree file modes for operation, such as VIRTUAL, SHARED, etc. This is a good place to look at **d-tree's** lookup (SCAN) features. Use the lookup key (?) while in the mode field on the screen. This will provide a list of valid file modes which looks something like this:

```
Sat Apr 28                          FairCom                          11:04:26
                                  Valid Codes
Select
   1      (FIXED   | VIRTUAL   | SHARED)
   2      (FIXED   | PERMANENT | EXCLUSIVE)
   3      (FIXED   | PERMANENT | SHARED)
   4      (VLENGTH | VIRTUAL   | EXCLUSIVE)
   5      (VLENGTH | VIRTUAL   | SHARED)
   6      (VLENGTH | PERMANENT | EXCLUSIVE)
   7      (VLENGTH | PERMANENT | SHARED)



                          Enter Desired Option:[__]
                             FairCom (c) 1988
                            Press ESC ESC to EXIT
```

This list provides all valid alternatives for the **c-tree** file modes. Components include: fixed versus variable length, permanent versus virtual, and shared versus exclusive. Simply select the combination you desire (in this case 5).

Back on the File Description screen, the size of each field along with alignment requirements will supply the record length. The catalog will count the number of indexes when information is posted, therefore these fields are protected.

Next, we can specify fields within the record. For each field, we can supply:

- the field name
- the field type
- the length of the field
- the number of decimal digits (for real or floating values)
- a description
- an indication of the first variable length field (if applicable)

Supply this field information from the specifications given above for the Vendor Master file. Let's make this a **variable length file**. The vendor code and balance will be in the fixed length portion of the file, and the rest is considered variable. To make this a variable length file simply key a "**VL**" in the "**First Vlen Field**" position for the first variable length field, in this case vendor name.

When you are done, your screen should look something like this:

```
Sat Apr 28                          FairCom                                   11:22:09
File Name:[vendor       ]     File Description: Vendor Master File
Version Number: 1.0           System Name:      Small Project Acct.
File Type: MASTER        Extension:  4096 Mode:  5 Rcd Len:  ___ Indexes: ___
 Field                     Field                                        First Vlen
 Name         Type Len Dec Description                                    Field
 vnd_code      A    11  _  Vendor Code                                     __
 vnd_bal       DF    8  2  Vendor Balance                                  __
 vnd_name      A    41  _  Vendor Name                                     VL
 vnd_addr1     A    41  _  Vendor Address 1                                __
 vnd_addr2     A    41  _  Vendor Address 2                                __
 vnd_city      A    21  _  Vendor City                                     __
 vnd_state     A     3  _  Vendor State                                    __
 vnd_zip       A    11  _  Vendor Zip                                      __
 vnd_type      A     5  _  Vendor Type                                     __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
 ____________ __ ___ _  ________________________________________    __
                              Press ESC ESC to EXIT
```

Now we can define the index structure by using the (F3) key. Pressing will display this screen:

```
Sat Apr 28                          FairCom                                   11:24:05
File Name: vendor             File Description: Vendor Master File
Version Number: 1.0           System Name:      Small Project Acct.
File Type: MASTER        Extension:  4096 Mode:  5 Rcd Len:  ___ Indexes: ___

                               Key Definitions

     Key             Key     Key  Dups  Null  Empty       Index File
     Name            Length  Type Ok    Key    Char  Mode Ext   Name
    [___________]    ____    __   N     N     __     __   _____ ____________

                    Field Name      Offset    Length    Mode

                    ____________    ____      ___       __
                    ____________    ____      ___       __
                    ____________    ____      ___       __
                    ____________    ____      ___       __
                    ____________    ____      ___       __
                    ____________    ____      ___       __
                    ____________    ____      ___       __

                              Press ESC ESC to EXIT
                                                                      Trn No:  1
```

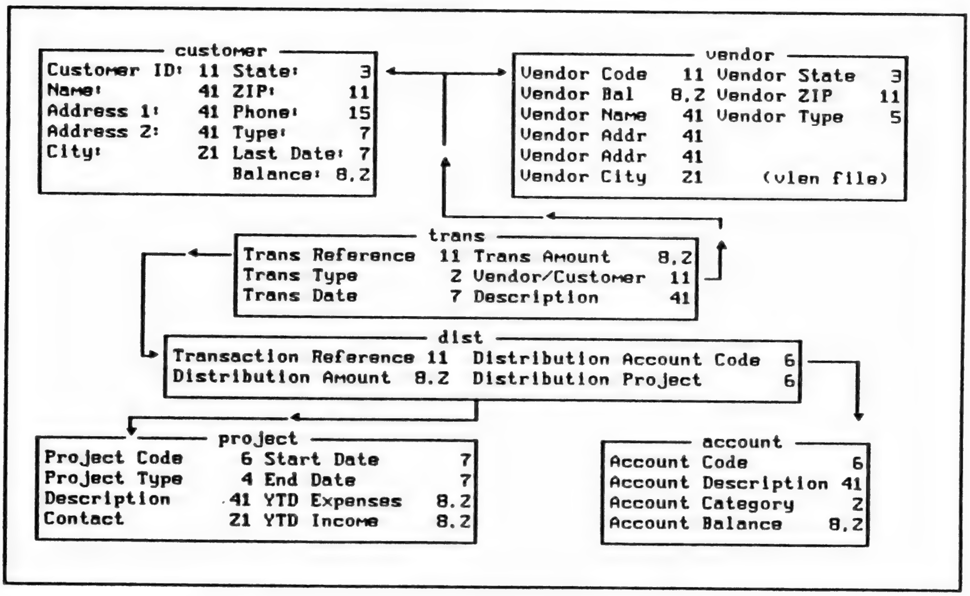
Enter a key name of "vnd_code_idx". You can use the DEFAULT key to move through the items until you reach the Field Name. Enter "vnd_code" to set the vendor code as the primary key. The offset is 0, the length 10, with mode of 0.

CAUTION: Developers who have worked with **c-tree** are use to entering the offset of a field as it pertains to the record structure. In **d-tree**, however, the catalog knows at what offset a field begins in relation to the record structure, without you telling it. Therfore, this offset is NOT what you are used to thinking of in **c-tree**. This is the **offset within the field** to start this key segment. Example: Let's say you had a date field that was a six byte string in the form MMDDYY and you wanted to build an index using this field. The first segment would have the date field name with an offset of 4 with a length of 2(selecting the YY), followed by a second segment with the same date field name with offset of 0 and length 4(selecting MMDD). Again, **THIS IS THE OFFSET WITHIN THE FIELD, NOT THE OFFSET WITHIN THE RECORD STRUCTURE**. After all key definition information is entered, use the POST key (END for DOS) to complete the entry. This will then return you to a blank File Definition screen. We are still in ADD mode so let's enter the last three file descriptions into the catalog.

Enter file file definitions for the following files:

- **Account Code File.**
- **Transaction File.**
- **Distribution File.**

Use the relationship chart below for field sizes. Be sure to set up the key structure for at least a key on the first field of each of these files before posting the file description. If you forgot, go back into View/Modify mode for each file and do it then. Note: The index for the "dist" file should support duplicates so many records related to a single "trans" record may be entered.



Once we have our file definitions in the catalog, we have a number of helpful and powerful functions available to us. Let's take a quick look at some of these.

Now enter the Data Dictionary in View/Modify mode. Select the Vendor file.

As seen earlier, Pressing Function Key 3 (F3) will display the index definition. In order to return to the File Description screen, simply press Function Key 8 (F8).

Other functions that we can activate from the File Description screen include:

- C record structure creation (F1)
- DODA structure creation (F2)
- Parameter file creation (F4)
- Incremental structure creation (F5)
- Instant file maintenance (F6)

Short examples of each of these follow.

F1 KEY- Creating C Record Structures: If we would like to generate a C record structure, simply press the (F1) key. The following screen should appear:

```
Sun Apr 28                          FairCom                                   13:03:25
File Name:[vendor       ]     File Description: Vendor Master File
Version Number: 1.0           System Name:      Small Project Acct.
File Type: MASTER        Extension:  4096 Mode:  5 Rcd Len:   20 Indexes: ___

 /* C Record Structure */
 struct UC0 {
 TEXT           vnd_code[11];            /* Vendor Code */
 double         vnd_bal;                 /* Vendor Balance */
 TEXT           vnd_name[41];            /* Vendor Name */
 TEXT           vnd_addr1[41];           /* Vendor Address 1 */
 TEXT           vnd_addr2[41];           /* Vendor Address 2 */
 TEXT           vnd_city[21];            /* Vendor City */
 TEXT           vnd_state[3];            /* Vendor State */
 TEXT           vnd_zip[11];             /* Vendor Zip */
 TEXT           vnd_type[5];             /* Vendor Type */
 } uc0[3];

                          Press ESC ESC to EXIT
```

Pressing the (F1) key a second time will allow you to dump this structure to a disk file. **d-tree** will request the name of the source file to which you want the specification written. You will also be asked if you wish to append these specifications to an already existing source file. If you respond "N", any current contents of the filename specified are overwritten with these specifications. Try this. This is a quick way to save you some keying when creating programs, although there are better ways, as you will soon see.

F2 - Create DODA Specs: Next, we can generate the DODA structure by pressing (F2). Those of you who are using r-tree are already familiar with the data object definition array (DODA). **d-tree** uses the same concept. The DODA is the table used to assign symbolic names to fields in order to use the fields in a script interface. See DODA section in the **d-tree** reference manual for a complete definition of the DODA. The (F2) key will produce the DODA definitions. Once again, pressing (F2) a second time will activate the **dump to disk** option. The same procedure is followed as before, supplying the source file name and whether to append or overwrite the file. The following screen shows how your screen will look after the second (F2) has been hit.

```
Sun Apr 28                              FairCom                              13:03:52
File Name:[vendor        ]      File Description: Vendor Master File
Version Number: 1.0             System Name:       Small Project Acct.
File Type: MASTER          Extension:  4096 Mode: . 5 Rcd Len:     20 Indexes: ___


 /* DODA Array */
 DATOBJ DTSDDODA[] = {
 {"vnd_code"     , NULL,RTSTRING ,11},     /* Vendor Code */
 {"vnd_bal"      , NULL,RTDFLOAT , sizeof(double) },/* Vendor Balance */
 {"vnd_name"     , NULL,RTSTRING ,41},     /* Vendor Name */
 {"vnd_addr1"    , NULL,RTSTRING ,41},     /* Vendor Address 1 */
 {"vnd_addr2"    , NULL,RTSTRING ,41},     /* Vendor Address 2 */
 {"vnd_city"     , NULL,RTSTRING ,21},     /* Vendor City */
 {"vnd_state"    , NULL,RTSTRING , 3},     /* Vendor State */
 {"vnd_zip"      , NULL,RTSTRING ,11},     /* Vendor Zip */
 {"vnd_type"     , NULL,RTSTRING , 5},     /* Vendor Type */
 {"","",0,0,-1}
 };


                              Press ESC ESC to EXIT
```

F3 - Maintain Index Definitions: The (F3) key has already been demonstrated. It takes us to the Key Definition screen.

F4 - Create Parameter File(optional maint pgm): Moving on to the (F4) key, we can generate the parameter file specifications based upon our Data and Key Definitions for this file. Hit the (F4) key. The Parameter file will be displayed. Pressing the (F4) key again will provide the option to write a parameter file to disk and ,optionally, create a parameter type maintenance program. Because we have not created a maintenance program for the vendor file yet, let's hit the (F4) key a second time. The following screen appears:

```
Sun Apr 28                          FairCom                                13:05:45
File Name: vendor             File Description: Vendor Master File
Version Number: 1.0           System Name:      Small Project Acct.
File Type: MASTER          Extension:  4096 Mode:  5 Rcd Len:    20 Indexes:   1

 /* Parameter File */
 10 1 4 2                                                <- Initialzation
 0 dummy.dat 128 0 3 0 dummy1 dummy2                     <- Dummy Lock File
 1 vendor.dat 20 4096 5 1 vnd_code vnd_type              <- Data File
 2 vendor.idx 11 0 0 0 4096 1 1 32 1 ven_code_idx        <- Index File
    0  11   0                                            <- Key Segment

                           Dump Specs to Disk
  This option will dump the file specifications you are viewing to disk.
  Key in the source file name of your choice WITHOUT an extention.
  The file name extension will default. A ".p" extention is used when
  viewing parameter files.  A ".h" extention is used for incremental
  structures. A 'Y' for the create option results in an additional ".c"
  source file to be created for a standard maintenance program.

 Name of Source File(s): vendor__ Do you want to Create Program Source:[Y_]
```

Give your source file the name "vendor", then create program source by responding with a yes ("Y"). A file named "vendor.p" will be created with or without the ("Y") response. With the ("Y") a program source file ("vendor.c"), a header file for the DODA definition ("vendor.h") as well as a d-tree script file ("vendor.dts") will be created.

The program to be produced will be entered into the program dictionary using the the program dictionary entry screen as shown below. Enter your vendor program into the program dictionary using this screen as a guide:

```
Sun Apr 28                          FairCom                             13:06:37
                               Program Dictionary


                  Add or Update Program to Code Dictionary

program name:          vendor
program version:       1.0___
program system name:   Small Project Accounting___
program description:   Vendor Master File Maintenance Program_
program type:          std maint______










                           Press ESC ESC to EXIT
    Program Specs Now Written. Do you want to Submit to Compile Que?.[Y_]
```

Once the posting is complete, **d-tree** generates the various specification files. Once these are written to disk, you are asked if you wish to submit the program to the compile queue. The **Compile Que** is a time saving device that allows you to submit progams which need to be compiled to a single list. The compiles can then be initiated all at one time from the Master Menu screen. More on this a little later. Respond yes ("Y") to this prompt. You will be ask to press return once the program has been submitted to the compile queue, which will the return you to the main menu.

F5 - Create Incremental File Structures (optional maint program): As with the (F4) option, the F5 option will allow for the creation of a maintenece file program. Lets pick another file for which we have no maintenece program yet. Go into the Change/View Mode of the data dictionary and select the ACCOUNT file. When viewing the account file definition press the (F5) key. the following screen will appear.

```
Sun Apr 28                                     FairCom                                    13:09:51
File Name:[account     ]     File Description: Account Code File
Version Number: 1.0          System Name:      Small Project Acct.
File Type: MASTER        Extension:  4096 Mode:  1 Rcd Len:    58 Indexes:   1


 ISEG DTSISEGS[] = { /* ISEGS */
 { 0, 6, 0   } ,
 };

 IIDX DTSIIDXS[] = { /* IIDXS */
 {
 6, /*key length*/
 0, /*key type*/
 0, /*duplicate flag*/
 1, /*null key flag*/
 32, /*empty character*/
 1, /*number of segments*/
 DTSISEGS+0, /*segment info*/
 "act_code_idx"   /*r-tree symbolic name*/
  } ,
 };
                              Press ESC ESC to EXIT
```

Use the rollup and rolldown keys to see entire structures. Follow the same procedures as described for the parameter file program to create a maintenance program that uses incremental structures, by pressing (F5) a second time. Enter a source file name of "account". Everthing will process as before, creating the appropriate source files. The difference is that there will be no parameter (".p") file and the header (".h") file will contain the incremental structures along with the DODA definiton. The #define in the mainline (".c") file for parameter files will not be included. (compare the source files for "vendor.*" to the "account.*" to see difference between parameter file maintenance programs and incremental file programs).

Now what about this compile queue? To understand this in detail, return to the operating system prompt and look at the contents of the file "DTCOMPIL.QUE". (this can be done with the "type" or "cat" command).This file contains the names of the programs ready to be compiled. During the installation of **d-tree**, a batch file called "DTQUE.BAT" was configured. When executed, this file calls the program "DT_DOQUE" which in turn reads each entry from "DTCOMPILE.QUE". It the submits each program name to the batch file that compiles the program (DTCOMPIL.BAT). Look at these files to understand the flow. Return to the catalog's main menu and select the option to "Execute the Compile Que". This will compile the two programs you created ("vendor" and "account") by calling DT_DOQUE.

F6 - Instant File Maintenance: One of the most useful options within the catalog is the ability to do instant maintenance over any file defined in the data dictionary. How many times have you wished you had a "*quick and dirty*" way to do raw maintenance for all data elements of a file (either to fix a corrupted flag, or add test data during development) without wasting valuable time. Go back into the data dictionary and select the account file. While viewing the file definiton hit (F6). The catalog will create a default script (much like the "**run**" program did) and begin executing its definiton. Add some records to the account file with option 1 and then View them with option 2. Escaping out will return you to the catalog. (Note: you never left the catalogs executable mainline. This is another example of using the powerful tools within d-tree to create data independent mainlines. When (F6) was selected, the catalog's definiton was freed from memory and the project definiton was parsed into memory. The same mainline processed the project file. When we escaped back out of project maintenance, its definition was freed from memory and the catalog definiton was re-loaded from the ability dictionary.)

F7 - Not Used

F8- Return to Definition Maintenance: This will return you to the primary file definition screen when viewing any other screen provoke by the function keys.

F9 - Delete line in subfile.

F10 - Insert line in subfile.

EXERCISE: (must be done to complete file definitons)
We now have all files, except one, in the data dictionary. The project file definiton we created with "**run**" has not yet been brought into the catalog. Return to the program dictionary and import the project definiton in the same manner as the customer definiton. (See start of this session.) After importing the project files into the Data Dictionary, it will be necessary to modify the field names in order to to distingush them from the fields that where defined in the customer file. *(If you recall, the customer master file that was imported also uses the symbolic identifiers F0001, F0002, F0003...)* Following sessions will create a DODA with both files defined. Non-unique names will cause the same symbolic field names for two different fields.

d-tree Tutorial - Session 4

2.4 Muti-File Program - Tutorial

Let's explore some other capabilities of the catalog. Now that each of our files are defined, we can create a Multiple-File File Maintenance Program. This would allow validating information across files, having subfiles of information related to a master file, and performing updates directly to related files.

In the Data Dictionary, choose the "Select File" option. The catalog then requests which source specs to create by displaying a screen like this:

```
 Mon Apr 28                           FairCom                          20:07:48
                                   System Catalog
                               Select of Group of Files



                   Select the Desired Source Specs to Create

                       [_] Create a Parameter File
                        _  Create C Source Structures
                        _  Create a Data Object Def Array (DODA)
                        _  Create Incremental Structures
                        _  Create a d-tree Script
                        _  Create a r-tree Script

                       Enter Source File Name: ________

          Do you want to make an entry into the Program Dictionary: __

                                FairCom (c) 1988
                              Press ESC ESC to EXIT
```

This screen allows you to specify which program components you wish to have **d-tree** generate for you. **d-tree** needs only the DODA, the Incremental Structures and the **d-tree** script. The parameter file, C source structures, and the r-tree script are optional. You select the specs you want by placing a "Y" in front of them. At the prompt for the Source File Name, enter the name you wish to call your new program (no extension). Next you are asked if you wish to enter this into the Program Dictionary. This will cause **d-tree** to create a ".c" mainline file as well as prompt for an entry into the Program Dictionary.

For our example, let's specify that we want all available source specs (remember, A Parameter File and Incremental Structures are really mutually exclusive, so select one or the other). Provide a Source File Name of "**posting**", for "Transaction Posting Program", and specify that **d-tree** should enter the specs into the Program Dictionary.

d-tree then allows us to specify which of our files in the data dictionary are to be included in our Multi-File program. It asks for either a file name, file description or file system. This is the same "PROMPT" that you see in the View/Modify Mode. It is easiest to respond to "PROMPT" screen with "0" to specify a scan of all files in the data dictionary.

The scan should list our files like this:

```
Mon Apr 28                          FairCom                              20:08:17
                                 Data Dictionary
Sel File Type    Name          Version Description                System
[_]MASTER        account         1.0    Account Code File          Small Project Ac
 _ MASTER        customer        1.0    customer                   IMPORTED
 _ MASTER        dist            1.0    Distribution File          Small Project Ac
 _ MASTER        project         1.0    project                    IMPORTED
 _ MASTER        trans           1.0    Transaction Master File    Small Project Ac
 _ MASTER        vendor          1.0    Vendor Master File         Small Project Ac









                               FairCom (c) 1988
                             Press ESC ESC to EXIT
```

The "Sel" column allows us to not only select which files we wish to include but also which order they will reside in the source code that is created (ie: placement in paramter file, incremental files, doda and scripts). We do this by entering a "1" for the first file we wish to use. This is normally the master file for this update. We select the next file as "2" and so on. At the same time, we need to change the"File Type" column. The file type is used to determine the kind of entries that will be created in the **d-tree** script. In other words how this file is going to be used in this program.

The following are valid entries for the type field:

- "**MASTER**" - Primary file that is to be maintained. Only one file that is selected should be designated as the master file.

- "**SFL**" - A subfile is a related group of records. A subfile is most commonly used in a one-to-many maintenance situation. (ie: an invoice master record with numerous line item detail records. These detail records would be maintained in a subfile.)

- "**VAL**" - Validation File. This file will be used to validate entries in the files that are being maintained. This file will be used as a "lookup" file allowing the user to scan thru this file, and select entries. (ie: When entering an invoice, the invoice record may require the customer number. The customer master file would then be defined as a "VAL" type file. This would allow validation of the customer number when it is keyed into the invoice, as well as allow a "lookup" into the customer master file for the user to select the proper customer.) Defining a file as a "VAL" type will cause script entries for the following: VALIDATE EDIT , SCAN and MAP.

-

Let's specify the Transaction File as our master file and the Distribution File as a subfile. Specify the Vendor File, the Customer File, the Account Code File, and the Project File for validation. Your screen should look like this:

```
Sat Apr 28                              FairCom                          06:52:25
                                    Data Dictionary
Sel File Type   Name         Version Description                System
 5 VAL          account        1.0    Account Code File          Small Project Ac
[4]VAL          customer       1.0    customer                   IMPORTED
 2 SFL          dist           1.0    Distribution File          Small Project Ac
 6 VAL          project        1.0    project                    IMPORTED
 1 MASTER       trans          1.0    Transaction Master File    Small Project Ac
 3 VAL          vendor         1.0    Vendor Master File         Small Project Ac




                                   FairCom (c) 1988
                                 Press ESC ESC to EXIT
```

Hit POST Key when Ready.

After the source specs are created the screen will prompt to the program dictionary entry: Enter data as shown on the following screen and submit this program to the compile queue.

```
Sat Apr 28                          FairCom                           06:54:08
                               Program Dictionary


                  Add or Update Program to Code Dictionary

program name:           posting
program version:        1.0___
program system name:    Small Project Accounting___
program description:    Transaction Primary Posting Program____
program type:           MAINT__________








                              Press ESC ESC to EXIT
      Program Specs Now Written. Do you want to Submit to Compile Que?.[Y_]
```

Now look at the specifications which we have generated:

- **"posting.c"** - program mainline.
- **"posting.h"** - program header (c structures, doda, incremental structures).
- **"posting.dts"** - d-tree script.
- **"posting.rts"** - r-tree script.
- **"posting.p"** - parameter file. (optional: only if parameter files were selected).

Return to the operating system prompt, and using your editor we will look at the files that were created. *(Press "ESC ESC" to back out of the catalog).*

"posting.c" - The ".c" file is very simple:

```
posting.c
#include "DT_DEFIN.H"
#include "posting.h"
#define   DT_DTS   "posting.dts"
#define   DT_RTREE
#define   MYIFIL 1
#define SFL
#include "DT_SCORE.c"
```

This is the mainline for our program. It includes necessary header files (including "posting.h"). It then defines the script file (**#define DT_DTS "posting.dts"**). Optionally you would also see a #define statement for the parameter file if it was selected. It finally includes "dt_score.c". This is **d-tree**'s standard mainline (also used by "run" program). The #define SFL indicates to "dt_score" that subfiles are involved. This mainline works with the other components we have created to actually perform the specified file maintenance.

> Note: **d-tree** is first and foremost a powerful toolbox of functions. The "run", "catalog", and programs created by the "catalog" are good examples of programs that can be written with the tools. The user who understands each tool and how they work together will get the most out of **d-tree**. As you will see in later sections of the manual, we strongly encourage looking at the "dt_score.c" mainline. It, again , is a good example of how to apply the tools in a dynamic manner.

"posting.h" - Next, look at the header file "posting.h". First, it contains the "C" structures for the six files. Next, is the DODA followed by the incremental structures (if selected) . Other entries which will be new to you are shown below. These entries are explained in detail in later sections. Simply note them for now. No changes are necessary at this time to these entries.

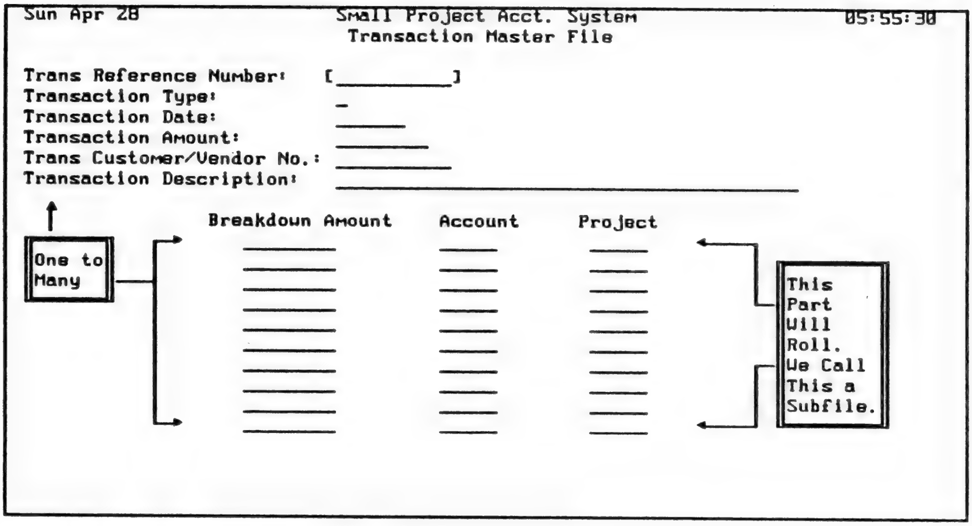
posting.h

```
DTTFUNCT DTSFUNCT[] = {                  User Defined Function Table
{ "DO_MAP",     DT_CALMP },
{ "",           DT_NULFP }
};

                                         Table of Hard Coded ADAMS
DTTHRDCD DTSHRD01[] = {
/* base pointer      , ref num    how many entries in definition table */
{ (TEXT *) DTSDDODA ,DTKDDODA , sizeof(DTSDDODA) / sizeof(DATOBJ)   },
{ (TEXT *) DTSISEGS ,DTKISEGS , sizeof(DTSISEGS) / sizeof(ISEG)     },
{ (TEXT *) DTSIIDXS ,DTKIIDXS , sizeof(DTSIIDXS) / sizeof(IIDX)     },
{ (TEXT *) DTSIFILS ,DTKIFILS , sizeof(DTSIFILS) / sizeof(IFIL)     },
{ (TEXT *) DTSFUNCT ,DTKFUNCT , sizeof(DTSFUNCT) / sizeof(DTTFUNCT) },
{ (TEXT *) 0        ,-1       , 0 }  /* terminaton Indicator */
};

#ifndef COMPILE
DTTHRDCD *DTSHRDCD[DTHRDCD] = {           Hard Coded Table Pointers
DTSHRD01,
};
#endif
```

"posting.dts" - the d-tree script. We must modify the **d-tree** script to specify exactly how we want the file interaction to occur. Let's look at it next. The first section looks very similar to the scripts we have reviewed before for the single-file maintenance programs. It pertains to the file which we designated as "MASTER", in our case the "trans" file. Look through this section. It terminates with the comment:

/*** SUBFILE INFORMATION ***/

Before we continue let's take a look at what our objective is for this program. We want a "trans" posting program, where we can break each transaction entry down by both the account number as well as the project. Each transaction can apply to more than one project, as well as more than one account. This breakdown is provided by entries in the "dist" file. The maintenance of this can be accomplished by providing a posting screen for the "trans" information, along with a scrollable portion of an entry screen which allows multiple ("dist") entries per transaction. In turn, we are about to modify the **d-tree** script so that our entry screen will look something like this when the program is executed:



Back to the script following the line:

/*** SUBFILE INFORMATION ***/

This second section describes the second file selected which was the "dist" File. As you recall, we specified this as a subfile of the "trans" File. You will notice that the first specification for this file looks like this:

```
posting.dts
/*** SUBFILE INFORMATION ***/

IMAGE(trans) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP} {BASE_ROW=??}



__________ ________ _____ _____

FIELD(trans)
/* Symbol Name   Input Attribute  Output Attribute  Input Order  I/O Special *

    dis_ref         NONE              NONE                2 /* Transaction Refern
    dis_amt         NONE              NONE                3 /* Transaction Amount
    dis_acct        NONE              NONE                4 /* Account Number */
    dis_proj        NONE              NONE                5 /* Project Number */

/*****/
```

This image (identified with the reference name "trans" - not related to our "trans" file name) represents the display format for the subfile records. Notice the question marks following the BASE_ROW parameter (base row is the first row on the screen that this image should start). Throughout the script file, **d-tree** left question marks to indicate values which we must supply before the script is useable. In this case, we must specify the row within the "master" image where the subfile display area (scrollable region) is to begin. First, let's go back and adjust our "master" image to that we can display this scrollable region in a sensible place. Modify the image for "master" to look like this:

```
posting.dts
IMAGE(master) {LSTFLD_ADVANCE}
@DATE                          Small Project Acct. System                 @TIME
                                Transaction Master File

Trans Reference Number:        __________
Transaction Type:              _
Transaction Date:              ______
Transaction Amount:            ________
Trans Customer/Vendor No.:     __________
Transaction Description:       ________________________________________

               Breakdown Amount         Account          Project

                 Clean up the "master" IMAGE by moving the fields
                 up and straightening the input lines. Add "dist"
                 column headings on line 11, leaving line 12 and
                 below blank. (Note: @DATE is considered to be on
                 line 1.)
```

We can then go back down to the IMAGE(trans) definition and specify a base row of 12 for the subfile display area. Note that on an IMAGE definition, line 1 is considered to be the first line below the IMAGE keyword. When a BASE_ROW is defined, it is simply an offset that is added to the lines as they are defined in the IMAGE section. You will want to delete the three blank lines which follow the IMAGE(trans) line, otherwise your scrollable region will start on line 15. *(BASE_ROW of 12. This image starts with three blank lines 12, 13, and 14; with fields starting on line 15.)*

As you will see below in the SFL_MAP section , when subfile records are written to disk, you have the option to MAP *(copy data from)* fields from the primary *(parent)* file ("trans") to the subordinate *(child)* file ("dist"). In our file definitions we have provided a means to relate these files by defining a transaction reference field in both files. As you will se below, when we write each "dist" record to disk, we will MAP the transaction reference number from the "trans" file to the "dist" file. In turn, we do not have to display (or maintain) the transaction reference number for the "dist" file on the screen. Modify the IMAGE(trans) and FIELDS(trans) sections as follows:

1) removing the two blank lines described above.
2) removing the transaction number.
3) adjusting the spacing so that the fields we are displaying will line up under the column headings defined on IMAGE(master). The result of these changes should look as follows:

```
posting.dts
/*** SUBFILE INFORMATION ***/


IMAGE(trans) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP} {BASE_ROW=12}

FIELD(trans)
/* Symbol Name   Input Attribute  Output Attribute  Input Order  I/O Special *

    dis_amt          NONE              NONE           1 /* Transaction Amount */
    dis_acct         NONE              NONE           2 /* Account Number */
    dis_proj         NONE              NONE           3 /* Project Number */

/*****/
```

The next section of the script defines the Subfile Specifications. This is what links the subfile (Distribution File) to the master file (Transactions), along with other requirements to process the subfile.

Again, notice that **d-tree** placed question marks where we need to provide values:

```
posting.dts
SUBFILE(trans)
SFL_IMAGE(trans) /* image to sfl record display(image that rolls)*/
SFL_RECORDS(??)  /* total number of records in subfile */
SFL_LINES(??)    /* total number of lines sfl takes up on screen  */
SFL_TITLE(?????) /* optional...image to be displayed as the title */

SFL_TARGET
/* key symbol name      fields for target      prefix */
    ???????                   ??????

SFL_MAP
/* parent field         child field             length */
    ?????                 ?????
    SFL_SEQ               ?????

SFL_MUSTHAVE
    ?????  /* fields that must exist to make a valid sfl rcd */
/****/
```

SUBFILE(trans) - First we give this subfile definition the reference name " trans" which our programs can use to refer to this subfile. Note that the word "trans" here has nothing to do with our file "trans". It is simply the reference name for this subfile.

SFL_IMAGE(trans) - Next we define the IMAGE that is going to be used as the scrollable region for this subfile. We have just completed our work on the "trans" image. This is the image we define to be used by the subfile.

SFL_RECORDS(??) - **d-tree** manages subfiles by two different methods.

- **1) MEMORY SUBFILES** - When the SFL_RECORDS keyword is given a value **other than one** (1), we are defining this subfile as a memory subfile. A memory subfile assumes that all information *(records from the detail file)* can be loaded into memory at one time. The size *(or number of detail records allowed)* of the subfile is restricted by the amount of memory available. Although this method does have this restriction, it does have advantages. **d-tree** can process a memory subfile much faster because all records are in memory. This method is best used when our application can define a limited number of detail records. In our case, we are breaking transactions down by "account" and "project". It is valid for us to assume that the user of this "posting" program will not need to "distribute" a single transaction between more that 50 "accounts" and "projects" *(in reality they will need only 5 to 10)*. In order to give them plenty of leeway, set the number of subfile records to 50 as follows:

  SFL_RECORDS(50)

- **2) UNLIMITED SUBFILES - d-tree** will manage paging subfile records on and off of disk, providing the capability to have an unlimited number of records in the subfile. *(NOTE: Limited only by physical disk space)*. This is done by simply defining the number of subfile records to be one (1) as follows:

  SFL_RECORDS(1)

  (NOTE: Keep this in mind later when you work with the "catalog" program. We use a subfile in that program to maintain the fields in a file. We ship this subfile as a fixed number, allowing 48 fields per file. You can increase this number if memory is available *(UNIX/XENIX ok....DOS is tight)* or change this to allow an unlimited number of fields per file by modifying the catalog script (dtcatlog.dts"). *The modification involves changing SFL_RECORDS(48) to SFL_RECORDS(1) in the SUBFILE(master) definition.)*

SFL_LINES(??) - Next we must specify the total number of lines the scrollable region is to occupy on the screen. This number should be evenly divisible by the number of lines used in the subfile image *(in our case 1 line)*. This will control how many records will be displayed in the scrollable region at one time.

Example: If our subfile image takes up three lines and we define the total number of lines to be nine, we will see three records on the screen at a time. We must also consider the number of subfile records we defined previously *(unless it is unlimited)*. If we define nine lines for the scrollable region, and each record takes up three lines, we need make the number of subfile record divisible by three , ie: 21.

Here are the rules:

- 1) The number of records in the subfile has to be divisible by the number of records per screen.
- 2) The number of records per screen is determined by the number of lines the subfile occupies on the screen (SFL_LINES) divided by the number of lines required by each subfile record. (number of lines in the subfile image-SFL_IMAGE).
- 3) The number of lines on the screen (SFL_LINES) must be divisible by the number of lines defined in the subfile image-(SFL_IMAGE).

In our case, our image ("trans") only takes up one line. Let's define SFL_LINES to be 10 as folows:
SFL_LINES(10)

We have already set SFL_RECORDS to 50, thus we will have ten (10) records per page in the subfile, and are able to roll (page up/page down) though five (5) pages of subfile records.

SFL_TITLE(????) - Subfile title (optional). This keyword provided a means to define a separate image which will automatically be displayed the first time the subfile is displayed on the screen. It is useful for subfile column headings when it is not appropriate to place these headings on the primary image. Because we have placed our headings on our primary image we have no use for this feature in this specific application. **Delete or comment out this line from the script.**

SFL_TARGET - Subfile Target. This keyword is used to define the connection between a subfile and a data base *(c-tree)* file. The SFL_TARGET definiton is made up of three parts:

- **Key Symbol Name** - Enter the symbolic name of the index which will be used to load this subfile. This will be the name of an index defined for the file from which we want to access records. In our case , we want to access records from the "dist" file. We have defined a key field (transaction reference number) in the "dist" file, as well as an index over this field. You provided a key symbol name when you defined this key in the catalog. If you do not remember this key name, either go back into the catalog and look it up, or even easier, this symbolic name is defined in the source specs which were created for this program. Look at the incremental file definitions in "posting.h" (or if a parameter file was created, look at "posting.p"). We willl assume the key was named "dist_ref_idx" for illustration purposes.

- **Fields for Target** - Enter the fields to be used to make up a "target" field that will be used in accessing the provided index when loading the subfile. **d-tree** automatically combines all fields and performs proper transformation on the fields in order to make up the "target". In our case, we will use the "transaction reference number" from the "trans" file. We want to load all records from the "dist" file that relate to a single entry in the "trans" file. The relation is done with the "transaction reference number". Enter the field names you used when you defined the "trans" file for the "transaction reference number". We will assume "trn_ref" for our illustrations.

- **Prefix** - The prefix is an optional entry. If entered it will be the first segment used when the "target" is formed. We will not use this feature in this application.

SFL_MAP - subfile map. The subfile map provides the capability to MAP *(copy)* data into subfile records as they are written to disk. Its most common use is to coordinate changes in the master file with related records in a subordinate file. For instance, if the user changes the "transaction reference number" in a record from the "trans" file , we would want the "transaction reference number" in each corresponding subfile record, to automatically be changed as well. We would therefore define that we want the "transaction reference number" from "trans" file to be **mapped** into the "transaction reference number" in the "dist" file. The "trans" file is considered the parent, and the "dist" file is the child. Enter the proper fields in your script. *(See the illustration on next page.)* We have assumed the following field names: "trn_ref" for "trans" file and "dis_ref" for "dist" file.

Some of you may have noticed a possible oversight in our design of the "dist" file. Because we are loading the subfile using a key that is solely defined for the "transaction reference number" (one key segment) , we have no guarantee that the subfile records will be consistently loaded in the same order each time. In otherwords, if we key in three subfile ("dist") records in a certain order for a specific "trans" record, and then load the subfile again, we **will** get the proper three records from the "dist file" , but they **may not be** in the same order that we keyed them. The solution to this is as follows: If we had defined a "sequence number" field in our "dist" file, we could then define the key used to load the subfile to be made up of two key segments (two fields). The first is the same as before, the "transaction reference number". The second segment (field) would be the "sequence number". If, as each subfile record was written to disk , a sequence number was **mapped** into this field, subsequential access to this file, using this index, would insure that the records were loaded in a consistent manner. The **mapping** of this sequence number into each record is obtained by using the special keyword "SFL_SEQ " as the parent field in the SFL_MAP section. The "sequence number" field is the child. We did not place a "sequence number" field into our "dist" file, so we will not use this feature now. Delete *(or comment out)* the line in the script where the SFL_SEQ has been defined. Remember this observation when you run the "posting" program.

SFL_MUSTHAVE - Subfile must have. The "SFL_MUSTHAVE" specification is used to identify any field or fields in the subfile record which must contain data before the record is considered to be a valid record. Simply list the field(s) that must contain data before this record is considered valid. In our case, we will not consider an entry unless there is an amount entered in the record. Thus, enter the field that you defined as the amount field in the "dist" file in this section. We will use "dis_amt" for our illustration.

The following illustrates the completed subfile section:

```
posting.dts
SUBFILE(trans)
SFL_IMAGE(trans) /* image to sfl record display(image that rolls)*/
SFL_RECORDS(50)  /* total number of records in subfile */
SFL_LINES(10)    /* total number of lines sfl takes up on screen  */
/* SFL_TITLE(?????)  optional...image to be displayed as the title */

SFL_TARGET
/* key symbol name      fields for target      prefix */
    dis_ref_idx             trn_ref

SFL_MAP
/* parent field         child field             length */
    trn_ref              dis_ref
/*  SFL_SEQ              ????? */

SFL_MUSTHAVE
    dis_amt   /* fields that must exist to make a valid sfl rcd */

/****/
```

The next and subsequent sections are identified with the comment:

/* VALIDATION FILE ***/**

When a file is selected as a "VAL" *(validation)* type, specifications are witten for the following **d-tree** keywords: EDITS, IMAGE, FIELD, SCAN, MAP. For every "VAL" file selected these specifications are repeated with the proper data for the file selected, but using the same keyword reference name. It is your responsibility to provide unique names for these keywords as shown below. For our discussion we will use the Account Code file. After completing the following steps to provide the "account" validation, repeat these steps for the other three files.

Step 1 - EDITS keyword. The ability to validate a field as a proper entry in another file is provided by the edit type "VALIDATE". Therefore we have created a default edit entry. You must complete this edit definition:

EDITS(validate) /* Place this edit where applicable */

Invalid ??? Error Text ???_fieldname VALIDATE ???_index valmap valscan

First, we need to ensure that the edit is defined in the proper EDITS section. The account number we are validating is located within IMAGE(trans). We do not have an EDITS(trans) section defined yet, so we must make one. Change the name from EDITS(validate) to EDITS(trans). If there already was an EDITS section associated to the applicable image, we would delete the EDITS(validate) keyword and move the default edit specs to the proper EDITS section.

The edit default specifications require the following information to be provided where **d-tree** placed question marks:

- specify an appropriate error message you wish to display if the edit fails.
- enter the symbolic name of the field to be edited. For our illustration we will use "dis_acct".
- enter the symbolic index name to be used for the validate. In this case it will be the key over the account file by account number. For our illustration we will use act_code_idx.
- change the associated map name (valmap) to a unique name. We will use "actmap".
- change the associated scan name (valscan) to a unique name. We will use "actscan".

The result of these changes should look as follows:

```
posting.dts
EDITS(trans)          /* Place this edit where applicable */
Invalid Account Number    dis_acct   VALIDATE   act_code_idx actmap actscan
```

You may prefer to move the EDITS(trans) section to be under the IMAGE(trans) and FIELD(trans) keywords so they they reside together in the script.

STEP 2 - MAP keyword. First change the reference name given to the MAP to the same name defined in the EDITS section. (ie: MAP(actmap)) Besides simply validating a field as a valid entry in another file, the "VALIDATE" edit type provides for two more features.

- 1) Allows the definition of a scan, by which the user can scroll through the valid entries in the associated file, and select (or optionally add) a valid entry. The scan is defined in step 3.
- 2) Once a valid entry is selected, the associated MAP is executed (actmap). This associated map typically defines data to be mapped (copied) from the selected record to any defined field(s). The one field we will want to specifically map is the field that we are validating. In our case, if an account record is selected, we will want the account number field from the "acct" file to be mapped into the account field in the "dist" file. This is achieved via the following map definition:

```
posting.dts
MAP(actmap)
/* source field          desination field    Map Type    length */
    act_code                 dis_acct          REPLACE
```

STEP 3 - SCAN , IMAGE, FIELDS keywords for validate. The last section for each validation file is the scan specifications as described earlier. The combination of the IMAGE, FIELD, and SCAN keywords should be familiar. Apply appropriate changes, specifically providing the unique reference name as defined in the edits. (actscan). Your changes should look similar to this:

```
posting.dts
IMAGE(actscan) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                        Small Project Acct. System                   @TIME
                              Transaction Master File
Select










                        Enter Desired Option: __
                           Press ESC ESC to EXIT
                             FairCom (c) 1988

FIELD(actscan)
/* Symbol Name    Input Atribute  Output Atribute  Input Order  Special */
    option           NONE             NONE            1

IMAGE(actscanroll) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP}



  __ ____ ______________________________________ _ ________

FIELD(actscanroll)
/* Symbol Name    Input Attribute  Output Attribute  Input Order  I/O Special *
    counter          NONE             NONE             1
    act_code         NONE             NONE             2 /* Account Code */
    act_desc         NONE             NONE             3 /* Account Descriptio
    act_cat          NONE             NONE             4 /* Account Catagory *
    act_bal          NONE             NONE             5 /* Account Balance */

SCAN(actscan) {IMAGE_OUT=actscan} {IMAGE_ROL=actscanroll} {IMAGE_INP=actscan}
```

BE SURE TO CHANGE ALL OCCURENCES OF THE NON-UNIQUE REFERENCE NAMES TO UNIQUE NAMES WITHIN EACH VALIDATION FILE SPECIFICATION!

Repeat these steps for the other validation file specifications.

If the changes are complete, you can compile the program "posting.c" (or if you submitted it to the compiler queue, simply select "Execute Compiler Q" from the catalog main menu.)

The program is now ready to run. Imagine how much work this process would have involved before **d-tree**!

posting.rts - If you select the r-tree option when you created source specs from the catalog, review the "posting.rts" file. What the "catalog" program did, is to simply dump the selected files one after the other into a default r-tree script. This script is not ready to run if more than one file was selected. What the "catalog" did do is save you alot of grunt work. The majority of the reports requirements are in the script. It's up to you to move things around to create your specific report. The following is a sample of what is dumped to the r-tree script:

posting.rts

```
START
/* VIRTUAL */
SEARCH
        FILE "trans.dat" ALL
SELECT
        ALL

/* CONTROL */
/* SORT */
/* ACCUMULATOR */

DISPLAY
     DEVICE         3
     PAGE_LENGTH    66
     SCREEN_LINES   24

IMAGE
PAGE_HDR
+
+ Date: @xxxxxxxx                                      Page: @xxx
        SYS_DATE                                       PAGE_NO
+
BODY

+----------------------------------------------------------

+

+  @XXXXXXXXXXX
   trn_ref
+  @XX
   trn_typ
+  @XXXXXXX
   trn_date
+  @99999999
   trn_amt
+  @XXXXXXXXXXX
   trn_vncust
+  @XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   trn_desc
PAUSE
```

EXERCISE:

Fine tune the single-file and multi-file maintenance scripts to make the Small Project Accounting Package more to your own liking.

d-tree Tutorial - Session 5

2.5 The r-tree Interface - Tutorial

To this point we have focussed on constructing maintenance programs. This session will deal with another necessity in developing a complete system - Reporting. In this session we will illustrate d-tree's ability to provide a "front end" interface into the report generation capabilities of **r-tree**. **r-tree** is FairCom's powerful report generator, providing search, select, sort, of data base information as well as a "report-painting" interface.

There is always more than one way to approach a problem. This tutorial illustrates but one technique. Let's start by explaining the general flow of this approach.

When a report is desired by the user, a "**report prompt screen**", presenting the fact that the specific report was selected, is displayed. Optionally the user is allowed to provide input which will have some control as to the output of the report. A "**base r-tree report script**" must exist providing the report definition. This should be a complete script which can be passed to **r-tree's "report" function** for execution with the following exception. Any definition lines in the VIRTUAL, SEARCH, SELECT, or SORT sections of the r-tree script that will change as a result of user input, should be "**left out**" of the script. The different variations of the "left out" definition lines are defined in a d-tree script with the RTREE ability. When the report is executed, the proper variations are "inserted" into the r-tree script. If input from the user will do nothing to change the report (no input fields defined on prompt) then the base r-tree script is a complete script.

The "report prompt screen" and the "inserted" lines are defined in a d-tree script with the IMAGE and RTREE abilities.

Once the user completes the "**report prompt screen**" entry process, the "**base r-tree report script**" is read one line at a time, writing each line to the "report work file". Before each line is actually written it is checked to see if is contains a r-tree keyword. If so, and this keyword is also defined in the d-tree script, the user's input is scrutinized to determine the proper definition variation line to be "inserted". This definition is then written to the "report work file". This work file is then used by the r-tree "report" function to execute the report.

In this approach we have split the logic up into two seperate programs: The first program will be refered to as the "**prompt program**". The second program will be refered to as the "**report program**".

- **Prompt Program:** The prompt program makes the call to d-tree's r-tree interface function DT_RTREE. This function will display the "prompt screen", check the input, and build the "report work file" (DTRTS.BAK). It then will call (turn control over to) the "report program".
- **Report program:** simply contains the call to r-tree's "report" function. The script to execute is the parameter passed to it by the "prompt program". The "report work file" name is this parameter.

To illustrate this technique let's first create a simple Customer Master Listing. Once we have it running we will add customer range ,select and sorting options. The nice part about d-tree is that the catalog program does the majority of the work.

Begin by running the catalog and clear the compile que. Now enter the Data Dictionary. The first things we must do is select the files we will be using to produce this report. Take the "Select File" option. Select the following items for d-tree to generate:

- **DODA specs:** Will create the doda needed by r-tree.
- **Incremental Structures or Parameter Files:** Used to open the files.
- **d-tree Script:** Create a default "prompt screen".
- **r-tree Script:** Create a default "r-tree report script".

Enter the name "**custlist**" to be used for the source, scripts, and executable files. Say (Y)es to the program dictionary option ,then enter a '**R**' for a "report type" program. The data dictionary data file prompt screen will then be displayed. Enter a '0' in the first field to present a file selection screen containing a list of all the available files in the data dictionary. Place a '1' in front of the Customer file then press the POST key. d-tree will then create the following files:

- **custlist.p** - Only if Parameter file is selected.
- **custlist.h** - containing the DODA and incremental structures (if selected).
- **custlist.c** - mainline for prompt and report programs.
- **custlist.dts** - default d-tree script for prompt screen.
- **custlist.rts** - default r-tree report script.

The program dictionary entry screen will then appear. An entry here will provide a program to data file cross-reference when running catalog reports. Enter som-thing like the following:

```
Mon Apr 28                           FairCom                                 17:09:56
                                Program Dictionary


                      Add or Update Program to Code Dictionary

program name:          custlist
program version:       1.0___
program system name:   Small Project Accounting___
program description:   Customer Master Listing________________
program type:          REPORT_________
```

Escape back to the main menu of the Catalog and execute the compile queue to compile our "prompt program". Once the compile is complete exit out of the catalog program, back to your operating system.

Let's take a look at the d-tree script that was created. Using your text editor, bring up the file "custlist.dts". A default prompt IMAGE has been created. Change this IMAGE to apear as follows:

```
IMAGE(report) {LSTFLD_ADVANCE}
@DATE                      Enter Your Report Prompt Title                    @TIME

**** ENTER YOUR REPORT PROMPT SCREEN HERE ****

FIELD(report)
/* Symbol Name    Input Atribute   Output Atribute   Input Order   Special */
   option           NONE                 NONE              1
```

Change This   To This

```
IMAGE(report) {LSTFLD_ADVANCE}
@DATE                      Customer Master Listing                           @TIME



                 You are about to run a Customer Master Listing

                                                              Note: we took
                                                              out the FIELD
                                                              Ability for we
                        Press ESC ESC to Cancel the Report    have no input
                                                              fields.
```

Next a report message IMAGE has been created. Change this screen to look like this:

```
IMAGE(message)


**** ENTER YOUR REPORT MESSAGE SCREEN HERE ****
@DATE          This Message Will Display When Report Starts Running     @TIME
```

Change This   To This

```
IMAGE(message)
@DATE                    Small Project Accounting System                  @TIME
                            Customer Master Listing





                      Report Now Running.... One Moment Please
```

This first report will be a simple listing, without any "run-criteria" entered by the user. Because we desire no criteria the "insert" definition lines that have been placed in the script are not necessary. Delete these lines as shown:

```
                              custlist.dts
RTREE(report)
USES_IMAGE(report)
USES_SCRIPT(custlist.rts)                               These Lines Stay Here
REPORT_PROGRAM(custlist.exe)
RUN_IMAGE(message)
CALL_TYPE(EXECL)
/* r-tree     criteria   Substitute */                  Delete from this line
/* keyword    fields     String     */                  Thru and including
                                                        This line............
   SEARCH     NONE   FILE "???.dat" ALL
              ???    FILE "???.dat" USING_KEY ???_KEY [  "{???}"
              ???    FILE "???.dat" USING_KEY ???_KEY    "{???}" ]
              ???
              ???    FILE "???.dat" USING_KEY ???_KEY [ "{???}" "{???}" ]

   SELECT     NONE        ALL

   VIRTUAL    NONE        LEAVE_OUT

   SORT       NONE        LEAVE_OUT
```

Our d-tree script is now ready. We have already compiled the "prompt program", which uses this script, from the compile que in the catalog. The catalog has also created a default r-tree script. This script is ready to run so for this example we will not modify it. You may want to view this r-tree script to see the work the catalog saved you. See the file "custlist.rts". We have one last necessity before we are ready to run the report. The "report program" must be compiled. Remember, the "report program" contains the r-tree "report" function and is called by the "prompt program". It just so happens that when we created the source for the "prompt program" from the catalog, we also created the source for the "report program". Both mainlines are defined in one ".c" file, which is controlled with #define. The catalog creates this ".c" with the "prompt program" definition set (#define RPT_PRMPT). In order to create the "report program" we must simply take out the #define RPT_PRMPT (comment it out in the code) as show below:

```
custlist.c
/* #define RPT_PRMPT */    <-- Comment out this #define
                               for "report program".
#ifdef RPT_PRMPT
/* r-tree report prompt program */
#include "dt_defin.h"
#define   DT_DTS  "custlist.dts"
#include "dt_dtrts.c"

#else

/* r-tree report execution program */
#include "dt_defin.h"
#include "dt_globl.h"
#include "dt_ddoda.h"
#include "custlist.h"
/* #define DTRETPGM  define return program */
#include "dt_dtrpt.c"
```

This "report program" has got to be compiled. In the d-tree script, were we painted the "prompt screen", there is a keyword in the RTREE section which defines the name of the "report program". This is the name of the program called by the "prompt program" (i.e.: REPORT_PROGRAM(custlist.exe). This name has been defaulted to the same name as the "prompt program". One or the other has to be changed. Do one of the following (but not both):

1) Rename the "prompt program" executable created by the compile que, and compile this ".c" file. You would then call the report using this "new name" which will prompt the user and then call the "report program" (which has the original name) to run the report. Remember this original "report program name" was defaulted in the d-tree script by REPORT_PROGRAM(custlist.exe)

2) When compiling the ".c" file, give the executable a different name. If this is done you must also go back into the d-tree script (custlist.dts) and change the report program definition. Change REPORT_PROGRAM(custlist.exe) to reflect your new name.

After they have completed compiling, test the report by executing the "prompt program". (Note: You may first wish to verify that you have data in your customer file to use in your test.) With minimal coding, we used d-tree to generate an r-tree report. d-tree takes a lot of the "grunt" work out of creating reports by placing the fields and formats in the r-tree script. The user is then expected to modify these scripts finalize the report. See your r-tree documentation.

Now lets add the ability for the user to add some "run-time" report criteria. Change the d-tree script "custlist.dts" to look as follows:

```
custlist.dts
IMAGE(report) {LSTFLD_ADVANCE}
@DATE                    Small Project Accounting System                  @TIME
                           Customer Master Listing

             From Customer Number: ______
             To   Customer Number: ______

             Select only the following State: __

             Sort Report in Zip Code Order: __
             (enter a 'Y' or leave blank)

             Send Output to Device: __ (default: 1, Printer #1)
             (1 = Printer #1: 2 = Printer #2: 3 = Screen: 4 = Disk File)

FIELD(report)
/* Symbol Name   Input Atribute  Output Atribute  Input Order  Special */
    opt1          NONE              NONE               1
    opt2          NONE              NONE               2
    opt3          NONE              NONE               3
    opt4          NONE              NONE               4
    opt5          NONE              NONE               5
```

Leave the message screen the same. Add "user criteria" with the d-tree RTREE ability as follows:

```
custlist.dts
RTREE(report)
USES_IMAGE(report)                           Remember: "Report Program" name
USES_SCRIPT(custlist.rts)
REPORT_PROGRAM(custlist.exe)
RUN_IMAGE(message)                           This is assumed to be the
CALL_TYPE(EXECL)                             index name over the customer
/* r-tree   criteria  Substitute */          file by customer number.
/* keyword  fields    String     */

SEARCH NONE  FILE "customer.dat" ALL
       opt1  FILE "customer.dat" USING_KEY customeridx [ "{opt1}"
       opt2  FILE "customer.dat" USING_KEY customeridx "{opt2}" ]
       opt1
       opt2  FILE "customer.dat" USING_KEY customeridx [ "{opt1}" "{opt2}" ]

SELECT  NONE       ALL
        opt3       cust_state={opt3}             Customer state field name

VIRTUAL NONE      dev INTZ 2 1
        opt5      dev INTZ 2 {opt5}

SORT    NONE       LEAVE_OUT                     Customer Zip code field name
        opt4       NO_MOD cust_zip
```

Change the r-tree script "custlist.rts" to look as follows. Note: we have commented out the lines where the instructions say "remove":

custlist.rts

```
START
VIRTUAL

SEARCH
     /* FILE "customer.dat" ALL  */

SELECT
     /*  ALL   */

/* CONTROL */

SORT

/* ACCUMULATOR */

DISPLAY
     DEVICE          dev
     PAGE_LENGTH     66
```

1. Un-Comment the VIRTUAL Keyword.
2. Remove this search line. It will be inserted at runtime from your definition in the d-tree script. Leave SEARCH keyword alone. It must remain in the r-tree script.
3. Remove this select line. It also will be inserted at runtime. Do not remove the SELECT keyword.
4. Un-Comment the SORT Keyword.
5. Change the DEVICE definition. Was DEVICE 3 ... Now DEVICE dev The d-tree script's VIRTUAL will control the output device.
6. Change the rest of the script as desired.

Run the report, and supply various search, select, and sort, criteria. Also note the use of the VIRTUAL keyword to control where the output is to be directed.

SUMMARY: The technique of splitting the reporting process into two different programs has the following benifits:

- 1) r-tree alone takes up a significant amount of code space. This allows a program to call a report, without the additional r-tree code residing in the program.
- 2) Because the "report program" processes the script that is passed as a parameter, we are able to create ONE "report program" which can handle an unlimited amount of reports, by simply changing the script that is passed to it (no need to compile over and over).
- 3) In a muti-tasking environment, the "report program" can be submitted to backgroud for processing, freeing up the user.

Refer to the RTREE ability in Section 7 for a further information.

THIS PAGE LEFT BLANK INTENTIONALLY

d-tree Tutorial - Session 6

2.6 Menus - Tutorial

From Session 1 of this tutorial through Session 5 we have learned how to use the d-tree toolbox to develop the many pieces of a Small Project Accounting System. We must now tie all these pieces together to enable the users to access each program easily. Traditionally this equates to using menus. Fortunately, the catalog provides a quick way to create menus. Follow these steps:

- 1) Enter the catalog and clear the compile que.
- 2) From the Data Dictionary Menu, select the "SELECT FILES" option. Although we need no files for menus, we will take advantage of this option's ability to create d-tree scripts and mainlines.
- 3) To Create a menu program called "menu" make entries on the select screen as follows:

```
Sat Apr 28                        FairCom                          13:47:43
                               System Catalog
                           Select of Group of Files


                  Select the Desired Source Specs to Create

                        _  Create a Parameter File
                        _  Create C Source Structures
                        _  Create a Data Object Def Array (DODA)
                        _  Create Incremental Structures
                        Y  Create a d-tree Script
                        _  Create a r-tree Script


                        Enter Source File Name: menu____

           Do you want to make an entry into the Program Dictionary: Y_
                          Type of Mainline to Create
                (M)aint Pgm : (R)-tree Report Pgm :  Men(U) Pgm
                          Enter (M,R,or U):[U_]
```

- 4) Once the d-tree script has been created, the catalog prompts for an entry into the program dictionary. Place an entry something like the following so this menu program will appear in the cross reference reports produced by the catalog.

```
Sat Apr 28                        FairCom                          13:49:02
                             Program Dictionary


                  Add or Update Program to Code Dictionary

program name:         menu
program version:      1.0___
program system name:  Small Project Accounting___
program description:  System Master Menu____________________
program type:         MENU__________
```

- 5) Now return to the catalog main menu and execute the compile queue. This will compile your menu.
- 6) We must now "**paint**" our menu to fit our needs. Edit the file "menu.dts". As you can see, the catalog has created a default d-tree script which looks like this:

```
menu.dts
IMAGE(menu) {LSTFLD_ADVANCE}
@DATE                          Enter Your Menu Title                        @TIME

**** ENTER YOUR MENU SCREEN HERE ****


FIELD(menu)
/* Symbol Name    Input Atribute   Output Atribute   Input Order   Special */
    option           NONE               NONE              1

MENU(menu)
 USES_IMAGE(menu)
/*     Call Criteria           Type of Call     Call Value */
   option=1  CURSOR=name          EXECL           my_program
   option=1  CURSOR=name          SYSTEM          my_program
   option=1  CURSOR=name          CALL            my_function
   option=1  CURSOR=name          RETURN          1
```

- 7) We will start by creating a simple menu. Change the script to look like this:

```
menu.dts
IMAGE(menu) {LSTFLD_ADVANCE}
@DATE                     Small Project Accounting System                @TIME
                               System Master Menu



         1. Customer Master Maintenance   4. Valid Account Codes Maintenance
         2. Project Master Maintenance    5. Post Transactions
         3. Vendor Master Maintenance     6. Report Menu


                          Select Desired Option: __



                               Press ESC ESC to EXIT
FIELD(menu)
/* Symbol Name    Input Atribute   Output Atribute   Input Order   Special */
    menu1            NUMERIC            NONE              1
MENU(menu)
 USES_IMAGE(menu)
/*     Call Criteria            Type of Call    Call Value */
       menu1=1                   EXECL        customex.exe  -Emenu
       menu1=2                   EXECL        project    -Emenu
       menu1=3                   EXECL        vendor     -Emenu
       menu1=4                   EXECL        account    -Emenu     -E pgm return
       menu1=5                   EXECL        posting    -Emenu       option.
       menu1=6                   MENUCALL     menu3
```

- Notice the attention to the **-Emenu** in the illustration above. One of the mainlines supplied with d-tree is "**dt_score.c**". This is the mainline used by most programs produced from the catalog. It is also the mainline for the "**run**" program. A parameter passed to this mainline logic with a -E (execute upon exit flag) is considered to be an executable file name that is to be called upon exit. This provides a means for the "called" programs to return the the "caller". Here we use it to return to the menu.
- 8) Save the d-tree script, and run the "menu" program.

We created a simple "traditional" looking menu. By simply changing our script we can create more creative menus, providing popup, pulldown, and nesting capabilities. Change your "menu.dts" script to look like the script below and re-run the menu program. **BUT WAIT**. You don't think we would make you do all this grunt work for the sake of an exercise. See the file "**ex_menus.dts**" on your disk. Use this file instead of doing all that editing. You could simply copy "ex_menus.dts" into "menu.dts" and then re-run the program. As you run the menu, study the script to understand the definition.

Before you study this script, a conceptual understaing of pop-up and pull-down menus is in order. d-tree regards **pop-up** and **pull-down** menus as simply another IMAGE. Because we want the cursor to travel between options, we have defined input fields on the image. These input fields are initilized with the "option's text" by means of the **DEFAULTS** ability. We do not want the user to be able to change this text, so we have defined the input attribute of **NOCHANGE** for the fields. In some menus, we want the user to be allowed to enter the first letter of an option in order to GOTO that option. In this case we use the **GOTO input attribute** instead of NOCHANGE. We want this field to be highlighted when the cursor enters this field. The output attribute of **INPUTRI** handles this aspect. Frame characters (+) help as visual aids. The menu call criteria uses the "CURSOR=field" notation to determine selection. The **HOOKS** ability allows for menus to pop-up when the cursor enters a menu selection. See the menus ability in section 7 for more information.

SPECIAL NOTE: As you may notice, input fields, used in this menu approach, are solely defined in the script without the use of the IFILS ability. They are not defined as hard coded variables in the program. The DODA containing these variables and their space is allocated at parse time, and is only done for the first IMAGE/FIELD definition in the script. Therefore, all input fields used in a menu script must be defined in the first IMAGE/FIELD definition, if you have not Hard coded a DODA or used the IFILS ability to define the DODA. Simply place the menu with the most input fields as your first IMAGE/FIELD definition in your script. This is only a limitation if variables are defined at runtime. If you hard code your own DODA or use the IFILS to define DODA, this does not apply. If this is confusing, define your DODA as hard coded or with IFILS ability.

ex_menus.dts

```
IMAGE(menu) {INPUT_ADVANCE=CR} {BACKGROUND=BROWN} {FORGROUND=BLACK}

          +Small Project Accounting
            @DATE        System Master Menu     @TIME

                         ________________
                         ________________
                         ________________

                                                 +
```

Frame. Title will be centered

Input fields for menu. Will be initilized with text by DEFAULTS ability below.

```
FIELD(menu)
 /* Symbol Name   Input Attribute  Output Attribute  Input Order */
    menu1          GOTO AUTO_HELP      INPUTRI            1
    menu2          GOTO AUTO_HELP      INPUTRI            2
    menu3          GOTO AUTO_HELP      INPUTRI            3
```

Allows first character of menu option to be entered to GOTO that menu selection.

Causes Associated HELP text to be displyed when cursor enters this field. Allows for tag along reference text for this type of menu. Notice message line when running menu.

```
DEFAULTS(menu)
 /* Symbol Name   Type of defaults      Defaults value */
    menu1         INIT                  Posting Menu
    menu2         INIT                  Report Menu
    menu3         INIT                  Other
```

Menu option text.

```
HELP(menu)
/* Help Text or Token          fields    */
Do Your Daily Posting          menu1
Goto Report Menu               menu2
Describe Other Here            menu3
```

Help text here is used as tag along text via the AUTO_HELP input attribute

```
MENU(menu)
 USES_IMAGE(menu)
/*   Call Criteria          Type of Call   Call Value */
     CURSOR=menu1           MENUCALL       menu2
     CURSOR=menu2           MENUCALL       menu3
     CURSOR=menu3           MENUCALL       menu4
```

Example: If cursor is on the "menu1" input field when return is hit then call the menu named "menu2".

```
IMAGE(menu2) {LSTFLD_ADVANCE}
@DATE                    Small Project Accounting System            @TIME
                              System Master Menu


        1. Customer Master Maintenance   4. Valid Account Codes Maintenance
        2. Project Master Maintenance    5. Post Transactions
        3. Vendor Master Maintenance     6. Report Menu

                      Select Desired Option: _
```

```
MENU(menu2)
 USES_IMAGE(menu2)
/*     Call Criteria            Type of Call   Call Value */
       menu1=1                   EXECL          customex.exe  -Emenu
       menu1=2                   EXECL          project       -Emenu
       menu1=3                   EXECL          vendor        -Emenu
       menu1=4                   EXECL          account       -Emenu
       menu1=5                   EXECL          posting       -Emenu
       menu1=6                   MENUCALL       menu3

IMAGE(menu3) {INPUT_ADVANCE=CR}  {BACKGROUND=BLUE} {FORGROUND=RED}
+
       ________________      ________________      ________________
                                                                        +



FIELD(menu3)
 /* Symbol Name    Input Attribute   Output Attribute   Input Order */
     menu1        NOCHANGE            INPUTRI          1
     menu2        NOCHANGE            INPUTRI          2
     menu3        NOCHANGE            INPUTRI          3

DEFAULTS(menu3)
 /* Symbol Name    Type of defaults       Defaults value */
     menu1         INIT                   Customer Listing
     menu2         INIT                   Other Reports
     menu3         INIT                   Query Data

HOOKS(menu3)
/* Hook Symbol Name             Condition                    Function Parameters    */
   BEFORE_INPUT  cur_image=menu3 AND cur_field=menu1  DT_IMGOT menu4
   AFTER_INPUT   cur_image=menu3 AND cur_field=menu1  DT_UNPOP menu4

   BEFORE_INPUT  cur_image=menu3 AND cur_field=menu2  DT_IMGOT menu5
   AFTER_INPUT   cur_image=menu3 AND cur_field=menu2  DT_UNPOP menu5
```

> Here we use the hook ability to provide POP_UP menus as the cursor enters a selection from "menu3". The first hook read as such: If the current image is "menu3" and the current field is "menu1" the do an image out function with image "menu4" before input, and after input unpop "menu4". We have not defined a hook for the third option on this menu. When the user selected this option, "menu6" acts a a PULL-DOWN menu.

```
MENU(menu3)
 USES_IMAGE(menu3)
/*    Call Criteria            Type of Call   Call Value */
      CURSOR=menu1             MENUCALL       menu4
      CURSOR=menu2             MENUCALL       menu5
      CURSOR=menu3             MENUCALL       menu6
```

```
IMAGE(menu4) {INPUT_ADVANCE=CR}  {BACKGROUND=BLUE} {FORGROUND=RED} {POP_UP}

                                                 Only display these sides of the
                                                 frame. The alternate frame type
      +      ____                                defines what the sides of the frame
             ____                                look like.
             ____
                    + {LEFT} {BOTTOM} {RIGHT} {FRAME_TYPE=3}
CONST(menu4)
 1 YELLOW BLACK

FIELD(menu4)
 /* Symbol Name   Input Attribute   Output Attribute   Input Order */
    menu1         NOCHANGE            INPUTRI           1
    menu2         NOCHANGE            INPUTRI           2
    menu3         NOCHANGE            INPUTRI           3

DEFAULTS(menu4)
 /* Symbol Name   Type of defaults      Defaults value */
    menu1         INIT                  PULL
    menu2         INIT                  THIS
    menu3         INIT                  DOWN


MENU(menu4)
 USES_IMAGE(menu4)
/*   Call Criteria           Type of Call   Call Value */
     CURSOR=menu1            RETURN         1
     CURSOR=menu2            RETURN         2
     CURSOR=menu3            RETURN         3

IMAGE(menu5) {INPUT_ADVANCE=CR}  {BACKGROUND=BLUE} {FORGROUND=RED} {POP_UP}


                          +   ______
                              ______
                              ______
                                        +
{LEFT} {BOTTOM} {RIGHT} {FRAME_TYPE=3}

CONST(menu5)
 1 YELLOW BLACK

FIELD(menu5)
 /* Symbol Name   Input Attribute   Output Attribute   Input Order */
```

```
    menu1         NOCHANGE            INPUTRI         1
    menu2         NOCHANGE            INPUTRI         2
    menu3         NOCHANGE            INPUTRI         3

DEFAULTS(menu5)
 /* Symbol Name    Type of defaults       Defaults value */
    menu1          INIT                   Report 1
    menu2          INIT                   Report 2
    menu3          INIT                   Report 3

MENU(menu5)
 USES_IMAGE(menu5)
/*    Call Criteria            Type of Call    Call Value */
      CURSOR=menu1             RETURN          1
      CURSOR=menu2             RETURN          2
      CURSOR=menu3             RETURN          3

IMAGE(menu6) {INPUT_ADVANCE=CR}  {BACKGROUND=BLUE} {FORGROUND=RED} {POP_UP}



                                                            ______      +

{LEFT} {BOTTOM} {RIGHT} {FRAME_TYPE=3}

CONST(menu6)
 1 YELLOW BLACK
FIELD(menu6)
 /* Symbol Name   Input Attribute  Output Attribute  Input Order */
    menu1         NOCHANGE            INPUTRI         1
    menu2         NOCHANGE            INPUTRI         2

DEFAULTS(menu6)
 /* Symbol Name    Type of defaults       Defaults value */
    menu1          INIT                   Query 1
    menu2          INIT                   Query 2

MENU(menu6)
 USES_IMAGE(menu6)
/*    Call Criteria            Type of Call    Call Value */
      CURSOR=menu1             RETURN          1
      CURSOR=menu2             RETURN          2
```

THIS PAGE IS LEFT BLANK INTENSIONALLY

THE CATALOG

3.1 CATALOG - Introduction

The catalog program is a powerful utility built using d-tree functions and abilities. The catalog (dtcatlog) provides data, index, program, relationships, and ability dictionaries which aid in the management of application development. Besides providing an "**instant" maintenance** capability for any data file defined in the data dictionary, the catalog provides a variety of productivity aids such as automatically **generating** C record structures, DODA **structures**, as well as "start-up" **r-tree and d-tree scripts**. This section acts as a "user's operation guide" to the catalog program while providing some technical insight. It assumes that the catalog program has been compiled, and the "validation file" (dt_catvd) used by the catalog has been loaded with the necessary codes. These codes are loaded when the import option is selected from the dt_catvd program as described in the COMPLETE THE INSTALLATION discussion in section 1.

Let's start with a technical look at the catalog. The catalog program is simply a maintenance program written with the d-tree tools. It maintains a specific group of c-tree files which we have called dictionaries. These dictionaries provide a place where we can store definition information for files, fields, indexs, index segments, programs, and abilities, as well as how each of these entities relate to each other. The illustration below presents these dictionaries. The code where these definitions can be found is in "dtcatdef.h" and "dtcatlog.h". Using typdefs to define these file layouts is intended to ease the chore of modifying to these definitions if the user cares to define additional information.

```
                    TABLE DICTIONARY (dt_cattd.dat)
                           File Information

typedef struct dt_cattd {
IFIL    td_def;                     /* file definition   */
TEXT    td_fil[DTCFILLN];           /* file name         */
TEXT    td_version[DTVERLEN];       /* version number    */
UCOUNT  td_colum;                   /* number of columns */
TEXT    td_desc[DTCDECLN];          /* file description  */
TEXT    td_system[DTCSYSLN];        /* system name       */
TEXT    td_flag[2];                 /* select flag       */
TEXT    td_type[DTCFILLN];          /* file type         */
         } DT_CATTD;
```

```
          COLUMN DICTIONARY (dt_catcd.txt)
                Field Information

typedef struct dt_catcd {
TEXT    cd_fil[DTCFILLN];       /* file name          */
TEXT    cd_version[DTVERLEN];  /* version number     */
UCOUNT  cd_fldseq;              /* field seq number   */
UCOUNT  cd_dec;                 /* number of decimals */
COUNT   cd_fldtyp;              /* field type         */
COUNT   cd_fldlen;              /* field length       */
TEXT    cd_fldnam[DTCFLDLN];    /* field name         */
TEXT    cd_inpatr[8];           /* field input attr   */
TEXT    cd_outatr[8];           /* field output attr  */
TEXT    cd_desc[DTCDECLN];      /* Field Description: */
TEXT    cd_dfalt[DTCDECLN];     /* Default Value:     */
TEXT    cd_vlen[DTCDECLN];      /* Var len fld id     */
        } DT_CATCD;

          INDEX DICTIONARY (dt_catid.txt)
                Index information

typedef struct dt_catid {
IIDX    id_def;                 /* index definition      */
TEXT    id_fil[DTCFILLN];       /* file name             */
TEXT    id_version[DTVERLEN];   /* file version          */
UCOUNT  id_seq;                 /* index sequence        */
TEXT    id_idxfil[DTCFILLN];    /* index file name       */
COUNT   id_members;             /* no of index members   */
COUNT   id_ifilmod;             /* index file mode       */
UCOUNT  id_ixtdsiz;             /* index file ext size   */
TEXT    id_idx[DTCIDXLN];       /* index symbolic name   */
        } DT_CATID;

          SEGMENT DICTIONARY (dt_catsd.dat)
              Index Segment Information

typedef struct dt_catsd {
ISEG    sd_def;                 /* index definition */
TEXT    sd_fil[DTCFILLN];       /* file name        */
TEXT    sd_version[DTVERLEN];   /* file version     */
UCOUNT  sd_seq;                 /* seg sequence     */
TEXT    sd_idx[DTCIDXLN];       /* index name       */
TEXT    sd_col[DTCFLDLN];       /* column name      */
        } DT_CATSD;

          PROGRAM DICTIONARY (dt_catpd.dat)
                Program Information

typedef struct dt_catpd {
TEXT    pd_pgmnam[DTCFILLN];    /* program name        */
TEXT    pd_version[DTVERLEN];   /* program version     */
TEXT    pd_system[DTCSYSLN];    /* program system name */
TEXT    pd_desc[DTCDECLN];      /* program description */
TEXT    pd_type[DTCFILLN];      /* program type        */
LONG    pd_stamp;               /* unused              */
COUNT   pd_status;              /* unused              */
        } DT_CATPD;
```

```
ABILITY DICTIONARY (dt_catad.dat)
Ability Information

This file is a c-tree variable length file where the entire record is
considered variable length (no fixed length portion).  It is used to
store the parsed representation of abilities as they appear in
memory. Ability definition stored in this dictionary can be "swapped"
into memory when required, eliminating the necessity of a parse.
Later in the documentation you will discover the power of "groups",
where ability definitions can be "grouped" together and store in a
file on disk. These definitions can then be "swapped" in and out of
memory. The ability dictionary can be looked at as the "group file"
on disk for the catalog program.


RELATE DICTIONARY (dt_catrd.dat)
Contains left and right pointers relating
entries in the other dictionaries.

typedef struct dt_catrd {
COUNT   rd_ldic;          /* dictionary for entry      */
POINTER rd_lrecno;        /* left side record pointer */
COUNT   rd_ltype;         /* left side record pointer */
COUNT   rd_rdic;          /* dictionary for entry      */
POINTER rd_rrecno;        /* left side record pointer */
COUNT   rd_rtype;         /* left side record pointer */
        } DT_CATRD;


VALIDATION DICTIONARY (dt_catvd.dat)

Simply a "look-up" or "validate" file used by the catalog
program to verify this such as file modes, field types, etc.


typedef struct dt_catvd {
TEXT   vd_major[3];         /* validation major code      */
COUNT  vd_minor;            /* validation minor code      */
TEXT   vd_desc[DTCDECLN];   /* validation description     */
COUNT  vd_count;            /* validation numberic field  */
TEXT   vd_misc[DTCFILLN];   /* validation misc field      */
       } DT_CATVD;
```

Taking a glimpse at the internal files definitions will help define some of the external operations of the catalog program. Let's execute the catalog program enter "dtcatlog" from the operating system prompt (shown here for DOS):

C> dtcatlog

The first time the catalog program is executed provokes the parsing of the d-tree script for the catalog program (dtcatlog.dts). Once this script is parsed, the definitions are stored in the ability dictionary, thus eliminating the necessity for parsing the next time this program is run.

The catalog contains multiple menus and options. The options where "user operation" is obvious will not be addressed to allow us to focus on major "operation issues" of the catalog.

3.2 CATALOG - Data Dictionary

The Data Dictionary maintains file, field and index definitions. The data dictionary menu provides a means to add, change and delete these definitions.
The following illustrates the data dictionary primary maintenance screen:

```
Thu May 26                         FairCom                            20:21:47
File Name:[___________]     File Description: ______________________________
Version Number: _____       System Name:      ____________________
File Type: ___________  Extension: _____  Mode: __  Rcd Len: _____  Indexes: ___
 Field                    Field                                     First Vlen
 Name        Type Len Dec Description                                 Field
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
 ___________ __   ___  _  ________________________________________     __
                            Press ESC ESC to EXIT
```

The following is a description of the fields on this entry screen. The screen in a combination fixed **IMAGE** followed by a "scrollable" **SUBFILE**. The fields maintained in this fixed image belong to the TABLE (or file) DICTIONARY (dt_cattd.dat). Some entry fields have default values defined in the catalog's d-tree script. Hitting the "default key" (shipped as the TAB character) will plug this default into the field. These fields are noted with "(default defined)" notation after the fields description.

- **File Name** - is the name of this file as is resides on disk. An extension is optional. d-tree will use the extension define for c-tree's incrementals if no extension is given. This extension can be found in the file "ctifil.h" in c-tree's directory which is typically set to ".dat".
- **File Description** - is a reference description to identify this file.
- **Version Number** - identifies which version of this file this definition represents. Multiple versions of the same file may be maintained simultaneously. (default defined)
- **System Name** - provides the means to group files by "system". Cross reference information "by system" may then be produced. (default defined)

- **File Type** - The File Type acts as a work field for d-tree at program creation time. Simply hit the default key (TAB key) to default to "MASTER" type of file. This field aids d-tree is determining the kind of d-tree script specifications to write when the user selects this file at program creation time. Valid types are:
  - MASTER - Primary file being maintained in the program;
  - SFL - File is maintained in a subfile in the program.
  - VAL - File is used to "validate" data in the program.

  See session 4 of the tutorial. This field will represent "how this file was used" the last time it was selected during program creation. (If this is confusing at this point in time, don't worry, it will become clear as you run the catalog). (default defined)
- **Extension** - The Extension represent the file extension length used by c-tree. See the c-tree documentaion for a complete description on file extension lengths. (default defined)
- **Mode** - identifies the file mode used by c-tree (i.e. FIXED | VIRTUAL | SHARED). See the c-tree documentation for a complete discussion on file modes. Valid file modes may be obtained by entering a question mark ("?"). (default defined)
- **Rcd Len** - The record length is a protected field , calculated by the catalog which displays the number of bytes necessary for each record.
- **Indexes** - identifies the number of indexes defined for this file, is also protected, and is maintained by the catalog program.

The following fields reside in the subfile portion of the screen and define the field definitions for the Column Dictionary (dt_catcd).

- **Field Name** - is the symbolic (variable) name for the field.
- **Type** - defines the type of field. Valid field types may be seen by entering a question mark ("?"). (default defined)
- **Len** - defines the length of the field.
- **Dec** - defines the number of decimal positions and pertains to floating field types only.
- **Field Description** - provides a reference description for the field.
- **First Vlen Field** - identifies the first variable length field in this file. Enter 'VL' (only on one field) if this file is to be a variable length file. This field will then be considered to be the start of the variable length portion of each record. See variable length file support in the c-tree documentaion for more information on variable length files.

3.3 Catalog - INDEX DEFINITIONS

Once the file and field information is entered, we can add the index definitions for the file. This is done by pressing F3 from this entry screen. The following index entry screen will appear:

```
                          Key Definitions

     Key              Key    Key  Dups  Null  Empty      Index File
     Name             Length Type Ok    Key   Char  Mode Ext  Name
    [customeridx ]    11     __   N     Y     32     1   4096 customer.idx

                Field Name      Offset    Length    Mode

 Index File     F0001           ______      11       __
 requirements   ___________     ______     ____      __
 Roll for       ___________     ______     ____      __
 next index.    ___________     ______     ____      __
                ___________     ______     ____      __    Key Segment
                ___________     ______     ____      __    information
                ___________     ______     ____      __    Roll for more
                                                           key segments.
```

Presented in this entry screen are two "scrollable" subfiles. The first, which is only displaying one subfile record at a time, defines the fields that make up the index definition. This is the information stored in the index dictionary (dt_catid.dat). Page-Up and Down will "scroll" this line allowing for additional index definitions. The following fields define the index:

- **Key Name** - Symbolic reference name for this index used in both d-tree and r-tree scripts.
- **Key Length** - Total Length of the key. This field is maintained for you by the program.
- **Key Type** - c-tree's key type which controls key aspects such as key compression and right-to-left-scan. Enter a (?) to select a valid key type. See the c-tree documentaion on key types for a further discussion. (default defined)
- **Dups OK**- (Y/N) indicates if duplicate key entries are allowed. Note when (Y)es is entered the catalog takes care of the extra four (4) bytes in the key length required by c-tree. (default defined)
- **Null Key** - (Y/N) indicates if an "index entry" , derived from a record, which is considered to be Null, should or should not be added to the index. The consideration to be Null is determined by the Empty Character definition below. (default defined)

- **Empty Char** - If a "index entry" is made up of entirely this character it will be considered a Null index entry. This is used in conjuction with the Null Key Flag described above. This character must be represented as a decimal value. Set this value to 32 (space) for d-tree maintained files for d-tree initialized fields to blanks. (default defined)
- **Mode** - Index file mode identifies the file mode used by c-tree (i.e. VIRTUAL | SHARED). See the c-tree documentation for a complete discussion on file modes. (default defined)
- **Index Ext** - The Index Extension represent the file extension length used by c-tree for this index file. See the c-tree documentaion for a complete description on file extension lengths. (default defined)
- **Index File Name** - is the name of the index file as it resides on the disk. The first index defined will always have an index file name. If the user leaves this field blank, the program will plug the name used for the data file with a ".idx" extension appended to the name. Each additional index may or may not have a disk file name. If an addition index definition is given a name, it will be treated as a seperate index file. This will restrict the use of this file definiton to be only used with parameter files. c-tree incremental file definition requires that all indexes reside in one physical file as members. Leaving the addition index disk file name blank will define this index as a member within the previous "parent" index. By "parent" we mean an index that did have a disk file name provided.

The second subfile on this entry screen is used to define the index segments that make up an index entry. This subfile is a "child" subfile, who's "parent" is the index definition subfile just descibed above. Notice when you scroll the index definition subfile, how it's child, the segment definition subfile, will scroll along with it. The "parent" subfile will not scroll when the "child" is scrolled. This illustates the power of d-tree's subfile routines when working with hierarchies of subfiles. The following is a description of each input field in the segment definition subfile:

- **Field Name** - the field symbolic name used to comprise this segment of the index. F8 will return you to the primary maintenance screen to see your field names. Then F3 again to return you to index definitions.
- **Offset** - The offset within the field to start the index segment. This **IS NOT the offset within the record** itself as most c-tree user's are use to. The field name gives the catalog the offset within the record. This offset provides the flexablity to defined pieces of a field as a key segment. Consider the following: Given a six (6) byte string which is storing a date in MMDDYY format, we want to build an index over this file by this date. Proper key segment definitions require the date field to be defined twice, as two seperate segments. The first date segment

has offset four (4) with a length of two (2), while the second date segment has an offset of zero (0) with a length of four (4). This will construct a key in YYMMDD format.

- **Length** - the number of bytes from the field to be used when constructing a key entry.
- **Mode** - segment mode define segment trasformation when the key entry is constructed. See c-tree's segment mode documentation for further discussing.

Once the index information is entered pressing the "post" key (END key in DOS) will post this definition to the applicable dictionaries.

3.4 Catalog - FUNCTION KEYS

SUPPORTED WHEN VIEWING A DATA FILE DEFINITION

- **F1 - C Structure** - d-tree will create a C data structure for the current file. Pressing the F1 key a second time provide a prompt to "dump" the created C data structure to a specified disk file.

- **Shift F1 - Import File Definition** from an existing C structure - While viewing the Add Data Definition screen, press the key combination Shift F1. A small pop-up window will be presented at the bottom of the screen.

```
=====================Import file definition from C Structure=====================
Source File:  [____________] First Field:  ____________
Struct Tag/No:  ____________  Last  Field:  ____________  Append Def(Y/N):  __
```

This feature will allow you to import data definitions from existing C structures residing in source files. You may import all or any portion of the C structure. You may overwrite the current file's data definition or append the imported information to it. The following is a description of each field presented on the Shift F1 pop-up window:

Source File - Enter the full name (including extension) of the C source file containing the data structure.

Struct Tag/No - Enter the structure tag name (token following the word 'struct') associated with the structure to be used or the occurence number of the structure as it is found in the source file. d-tree determines if the value entered is a tag name or occurence number by performing an ASCII to integer conversion on the entry. If the resulting value is greater than 0, the entry is considered to be a structure number otherwise, it is considered to be a tag name.

First Field - (optional) Enter the first field within the structure to be retrieved.

Last Field - (optional) Enter the last field within the structure to be retrieved.

Append Def (Y/N) - Should the new data definition information retrieved from this C source structure be appended to the current file definition displayed on the screen or should this information replace the current definition? Enter (Y)es to append it to the current definition or (N)o to overwrite the current definition.

- **F2 - Create DODA** - d-tree will create a DODA for the current file. Hitting F2 again will provide a prompt to "dump" this definition to disk.
- **Shift F2** - Import from DODA Source Specs - This function displays a similar screen to the one shown above for the SHIFT F1 option. This function provides the same ability except allows this import of data file definition to come from an existing DODA.
- **F4 - Parameter File** - create a c-tree parameter file for the current data file.
- **F4 F4 - Dump Parameter File To Disk** - Generate Maintenance Program Pressing F4 a second time presents the following pop-up menu prompting for the necessary information to "dump" the parameter file just created to disk and build a standard Add/Change/Delete/Print maintenance program over the current data file using the newly created parameter file specifications.

```
                            Dump Specs to Disk
 This option will dump the file specifications you are viewing to disk.
 Key in the source file name of your choice WITHOUT an extension.
 The file name extension will default. A ".p" extention is used when
 viewing parameter files.  A ".h" extension is used for incremental
 structures. A 'Y' for the create option results in an additional ".c"
 source file to be created for a standard maintenance program.

Name of Source File(s):[________]Do you want to Create Program Source: __
```

Source Filename - The source filename field is the file which will contain the parameter file data. DO NOT specify a file extension. The file extension '.p' will be assigned.

Create Program Source - A reply of 'Y' in this field will prompt d-tree to create a standard Add/Change/Delete maintenance program using the parameter file being dumped to disk. Both the C source and d-tree script are generated.

- **F5 Incremental Structures -** d-tree will create a c-tree incremental structure for the current file.

- **F5 F5 - Dump Incremental Structure to Disk -** Generate Maintenance Program Pressing F5 a second time presents a pop-up menu prompting for the necessary information to build a standard Add/Change/Delete/Print maintenance program over the current data file using c-tree's incremental file approach. This prompt acts the same as the prompt described above for F4 F4.

- **F6 - Instant Maintenance -** the catalog provides "instant maintenance" over any file defined in the data dictionary. Simply hit the F6 key while viewing a file definition and the catalog program will build a default d-tree script for this file. It will then parse this script, converting the catalog mainline into a standard ADD/CHANGE/DELETE maintenance program for the file selected.

- **F7 - Unused**

- **F8 - Return to Data Definition Screen** from other screens or windows presented when other function keys are pressed.

- **F9 - Delete A Subfile Record -** By pressing F9 when the cursor is positioned in a subfile record will delete that record from the subfile.

- **F10 - Insert a record into a subfile -** pressing F10 when positioned within a subfile will cause a blank subfile record to be inserted before the current field definition entry. All subsequent subfile records will be shifted down one line.

3.5 View/Modify Data Dictionary Definition -

The View/Modify selection provides the capability of randomly viewing and/or editing a file definition. After selecting the View/Modify selection d-tree presents a prompt screen containing three possible means of selecting a Data Dictionary entry.

```
Thu May 26                          FairCom                                20:22:23
                                Data Dictionary



Enter File Name:    [______________]
or
Enter File Desc:     ____________________________
or
Enter File System:   ____________________________
```

From this prompt the user can proceed to select the file desired.

```
Thu Jul 21                          FairCom                                14:21:40
                                Data Dictionary
Sel Name          Version Description                    System
  1 account        1.0    Account Code File              Small Project Acct.
  2 customer       1.0    customer                       IMPORTED
  3 dist           1.0    Distribution File              Small Project Acct.
  4 project        1.0    project                        IMPORTED
  5 trans          1.0    Transaction Master File        Small Project Acct.
  6 vendor         1.0    Vendor Master File             Small Project Acct.



                          Enter Desired Option:[__]

                            Press ESC ESC to EXIT
```

If the users makes changes to the file definition, the catalog checks to see if any critical information related to a file's definition (i.e. filename, field lengths, version, etc) has been changed. If so, the following prompt is displayed:

```
        File Definition Has Been Changed

  Is this a (N)ew version of the file ?
  If so the old version will be preserved.
                  or
  Does this (R)eplace the old definition ?
  If so old definition WILL BE LOST.

         (N)ew or (R)eplace :[__]
```

The user has the option to either make a new file definition while retaining the old or to replace the old definition with the new. Using this feature, multiple versions of the same file may be maintained simultaneously. This could prove to be useful in supporting a commercial application with multiple releases. If (N)ew is entered a new version of the file is created, provided a new version number has been entered. If (R)eplace is entered, the following screen will be presented:

```
Tue Jul 26                       FairCom                          09:05:59
                         File Reformatting Utility


             The Old File Definition is Will Be REPLACED.



        The file format facility requires both the old and the new file
        layouts. If you want to take advantage of the file reformat
        facility one of the following actions must be selected:

       [_]Reformat the following file in place.

        _ Create a stand alone executable to be used for
          reformatting at a later time. Program Name: ____________
                                        Old File Name: ____________
                                        New File Name: ____________

                   Place a 'Y' in desired option(s) then
                        Hit POST key to continue.
                         Copyright 1988 FairCom
                      Press ESC ESC to CANCEL REPLACE
```

Because the old file definition is being replaced, the user is given two reformatting options. File reformatting requires both the old and the new file definitions. Because the old file definition will be lost by this replace option the following options are provided:

- **Reformat in place** - This will cause the catalog to look in the current directory for the file for which you just made the change. If found , will reformat the file in place, followed by an index rebuild. By reformat we mean will physically move the existing data in the file to fit the new file definition.
- **Generate a stand-alone reformat program** which may be executed at a later time. This method is very useful if you do not have immediate access to the data file to be converted. This program can also be used to send to your customer along with your update to allow them to reformat their data files.

3.6 Catalog - Reformat

```
Mon Apr 28                      FairCom                       20:16:07
                             System Catalog


   Master Menu
   Data Dictionary
   Program Dictionary         <
   Relate Dictionary                  Add
   Execute Compile Que                View/Modify
   Clear Compile Que                  Delete
   Catalog Reports                    Reformat
   Return to OS                       Select File
                                      Text Out
                                      Text In


                           FairCom (c) 1988
```

The Reformat selection provides a way to reformat between two versions of the same file define in the data dictionary. After entering the file name from the prompt screen the catalog will present the files on a select screen shown below:

```
Wed Jul 27                            FairCom                              09:43:58
                              File Reformatting Utility
Sel File Type    Name         Version Description               System
 _ VAL           account        1.0   Account Code File          Small Project Ac
 _ VAL           account        2.1   Account Code File          Small Project Ac
 F VAL           customer       1.0   customer                   IMPORTED
[T]VAL           customer       2.0   customer                   SPAS
 _ SFL           dist           1.0   Distribution File          Small Project Ac
 _ VAL           project        1.0   project                    IMPORTED
 _ VAL           project        2.0   project                    IMPORTED
 _ MASTER        trans          1.0   Transaction Master File    Small Project Ac
 _ MASTER        ucs_invn.dat   1.0   UCS Inventory Master       UCS PROCESSING
 _ VAL           vendor         1.0   Vendor Master File         Small Project Ac

                          From File Entry

                          To File Entry
```

Entering an 'F' in front of the "FROM" file an a 'T' in front of the "TO" file. Press the 'POST' key (END key in DOS). A comparison of the two files is done, follwed by the reformat.

3.7 Catalog - Select File

```
Mon Apr 28                    FairCom                         20:16:07
                          System Catalog


 ═Master Menu══════
 Data Dictionary       <-----
 Program Dictionary            Add
 Relate Dictionary             View/Modify
 Execute Compile Que           Delete
 Clear Compile Que             Reformat
 Catalog Reports               Select File
 Return to OS                  Text Out
                               Text In



                        FairCom (c) 1988
```

The select files option provides a means to create a variety of source specifications used in d-tree, c-tree and r-tree development. It major advantage is that it allows you to select many files at a time. Once this option is selected the following screen is presented:

```
Thu May 26                    FairCom                         21:46:44
                          System Catalog
                      Select a Group of Files



              Select the Desired Source Specs to Create

                  [_] Create a Parameter File
                   _  Create C Source Structures
                   _  Create a Data Object Def Array (DODA)
                   _  Create Incremental Structures
                   _  Create a d-tree Script
                   _  Create a r-tree Script

                    Enter Source File Name: ________

         Do you want to make an entry into the Program Dictionary: __

                        Press ESC ESC to EXIT
```

To select any of the options on the Select File screen, type a 'Y' in the blank preceding the option.

- **Parameter File** - will build a c-tree parameter file.
- **C Source Structures** - will build the C language data structures.
- **Data Object Definition Array (DODA)** - will build a DODA structure.
- **Incremental Structures** - will build the c-tree incremental data structures.
- **d-tree Script** - will build a d-tree script with various ability definition sections depending upon the purpose of each file selected.
- **r-tree Script** - will build a default r-tree script.
- **Source File Name** - will be the name used by d-tree for the files which shall contain the generated source information. Each file will be uniquely identified by its related file extension. For example, Parameter files '.p', C source files '.c', d-tree script files '.dts' and r-tree script files '.rts'.
- **Program Dictionary-** An entry of 'Y' will direct d-tree to prompt for an entry into the Program Dictionary. It will also cause a ".c" mainline routine to be created based on the program type entered below.
- **Program Type** - Enter one of the valid program types to tell the catalog the type of mainline to create.

After completing entry on this screen press the 'POST' key. d- tree will respond by displaying a file selection screen. To select multiple files to be used in the source and script files, place a number or letter in the adjacent 'Sel' field which corresponds to the order in which you would have the files appear. This selection process is illustrated below.

```
Wed Jul 27                                  FairCom                          09:52:20
                                       Data Dictionary
Sel File Type    Name           Version Description                System
 _  VAL          account         1.0    Account Code File          Small Project Ac
 2  VAL          account         2.1    Account Code File          Small Project Ac
 _  VAL          customer        1.0    customer                   IMPORTED
 1  MASTER       customer        2.0    customer                   SPAS
 _  SFL          dist            1.0    Distribution File          Small Project Ac
 _  VAL          project         1.0    project                    IMPORTED
 3  VAL          project         2.0    project                    IMPORTED
 4  VAL          trans           1.0    Transaction Master File    Small Project Ac
[_]MASTER        ucs_invn.dat    1.0    UCS Inventory Master       UCS PROCESSING
 _  VAL          vendor          1.0    Vendor Master File         Small Project Ac
```

For purposes of creating the d-tree script you may also specify how each file is intended to be used in the program. Valid entries are MASTER (primary file to be maintained), SFL (subfile), and VAL (validate). This will control the specification written in the d-tree script. A full example of running this option is illustrated in session 4 of the tutorial.

3.8 Text Out/Text Out

The "Text Out" selection will dump the data in the catalog dictionaries to corresponding ASCII files. These files have the same names as the data files execpt with a ".txt" extension (i.e. dt_cattd.txt). The "Text In" selection will load all the dictionary files from their corresponding ASCII files. An excellent application of the "Text Out" and "Text In" selections would be to use them to port the dictionary data from one operating system environment to another where the binary data files are not compatible. (Such as from an MS-DOS based machine to a UNIX environment.)

3.9 Program Dictionary

The program dictionary options allow for standard maintenance over the dictionary file dt_catpd.dat. This file is used as a program reference file used by the catalog reports for cross reference information. Maintenance options allow for ADD/CHANGE/DELETE capability to program definitions. Entries are made into this file for programs that are created through the catalog.

IMPORT PGM - The import program option is provided to import file definitions that have been prototyped with d-tree's "run" program.
THIS OPTION IS ONLY INTENDED FOR d-tree SCRIPTS THAT WERE CREATED BY THE "RUN" PROGRAM. See session 3 in the tutorial for illustation of this import capability. Enter the filename of an existing d-tree script. The file extension ".dts" is assumed.

3.10 Relate Dictionary

The purpose of the Relate Dictionary is to maintain relationship definitions between programs, data files and d-tree abilities. The current version of the Catalog only permits viewing entries from this menu selection. The concept of relate maintenance will evolve in future realeases of d-tree.

3.11 Compile Que

d-tree provides a handy feature within the catalog which permits the user to accumlate programs to be compiled at one time. The compile que is nothing more that a text file with a list of program names. When a program is submitted to the compile que, it's name is simply appened to this file. When the execute compile que option is selected the program called dt_doque is called which reads this que file and compiles each program name found. The header file dt_compl.h shown below defines elements used by this process:

```
                              dt_compl.h
/* batch file compile definitions */
#define   DTCOMPILE "compile.bat"      /* program compile batch file */
#define   DTCATQUE  "dtcompil.que"     /* compile que file */
#define   DTPQNAME  "dt_doque.exe"     /* process compile que program name */
#define   DTCOMP_P  "dtcomp_p.bat"     /* batch file for "run" pgm -c compile
                                          option-parameter files pgm */
#define   DTCOMP_I  "dtcomp_i.bat"     /* batch file for "run" pgm -c compile
                                          option-incremental files pgm */
```

Clear Compile Que - this option simply deletes the text file defined above that is used as the compile que.

3.12 Catalog Reports -

The catalog report option displays various reports that are available from information posted to the dictionary. Run each report once you have data in you catalog.

THIS PAGE LEFT BLANK INTENTIONALLY

Applying the Tools

4.1 Basic Interpreted Screen I/O

You have seen some examples of productivity provided by **d-tree** while creating applications using "**run**" and the "**catalog**" program in the tutorial. The true **power of d-tree** lies within the toolbox of functions provided to build applications. This is made evident through the fact that the bulk of this manual is the description of individual functions.

The developer who discovers the power and flexibility of the **d-tree** tools, by efficiently assembling them together, will get the most out of **d-tree**. Both the "**run**" and "**catalog**" programs are examples of dynamic modules, being redefined at runtime, which were built using the tools.

The purpose of this section is to instruct you in the requirements and guide you in the construction of useful function calls. To illustrate these requirements and concepts we will be working with a series of example mainline modules. Each example module will illustrate functions which provide various capabilities. The first example module will outline the minimum requirements, while successive modules will present more in-depth examples of time-saving tools.

ADAM - Before we actually proceed through each example we must first establish the definition of a new term, **Ability Definition Allocated Memory** block (**ADAM**). A d-tree ADAM is, as its name describes, an allocated block of memory, static or dynamic, which contains definition utilized to perform abilities. An **ABILITY** is an activity performed by a program for a certain purpose. As you will discover, the user has the means to add his own abilities to **d-tree**. For example, the ability to screen I/O, interpret keystrokes, or manipulate data, may all be considered abilities. In order for the user's program to perform an ability, there is one or more functions that must be called relating to that ability. These object definitons (also referred to as Ability Definitions) are contained in ADAMs.

Let's take an example: We need the **ability** to project a screen out to the user. In order to project the screen we will need its definiton. By defining a **typedef** as a structure, we can define the fields within the structure that we would need to define this screen (coordinates, attributes, etc.). Then, given an allocated memory block that has been initialized with the screen's definiton (the IMAGE ADAM), we can provide functions to perform the screen output, passing a pointer to the definition structure, giving the function its required definition.

d-tree ABILITY - The term ability can be used in many contexts. For clarity we will outline what constitutes a **d-tree** "Ability".

- An "Ability" is an activity to be performed by a program for a particular purpose. (ie: screen I/O)
- An "Ability" is given a reference name. (ie: IMAGE).
- An "Ability" is assigned a reference number in DT_DEFIN.H. (ie: DTKIMAGE)
- An "Ability" definition typedef is defined in DT_TYPDF.H to contain the definition of objects necessary to perform the ability. (ie: DTTIMAGE)
- An "Ability" requires an initialized ADAM containing definitions of its related objects
- An "Ability" has a d-tree script interface syntax definition. (ie: IMAGE(master))
- An "Ability" has a parsing function which is used to convert its script definition into an ADAM. (ie: DTPIMAGE)
- An "Ability" has related function calls to perform its objectives which use a pointer to the ability's ADAM for required definition. (ie: DT_IMAGE, DT_IMGOT, DT_IMGIN).

There are at least three (3) ways to create an ADAM, or in other words, to initialize the allocated memory with the "Ability" definition.

- **Interpretive**
  The "Ability" definition, in script form, may be initialized into allocated memory by a **parsing function**. Let's use the keyboard "Ability" as an example. A terminal definition, as in the "termcap" file, is considered to be in **d-tree script form** defining your terminal's specific characteristics. When the keyboard's parsing function (DT_KEYBD) is called, memory is dynamically allocated and initialized with the keyboard defintion. This allocated block of definition is referred to as the keyboard "Ability"'s ADAM.

- **Hard Code**
  The "Ability"'s definition may be hard coded into the source code, either in the mainline or included in a header file, and compiled into the executable. Static memory is allocated then initialized with the ability definition at run time. This method restricts changing "Ability" definition somewhat, although it does provide smaller executable code as well as faster startup time, because the parsing functions are not necessary in the code and are not executed.

- **Dictionary Initialization**
  An ADAM can be stored in a c-tree variable length data file just as they are used in the catalog program. This data file is referred to as an (ADAM) Dictionary (the ability dictionary is simply an ADAM dictionary used by the catalog program). The group functions (DT_GPOUT and DT_GPINN) are powerful ways of swapping ADAM definitions in and out of memory. See group functions for more detail.

Let's continue with our first example and take a closer look at initializing ADAMs via the interpreted method.

NOTE: It is recommended that you either print or view the files used in the following discussions.

EX_IMAGE

The first example mainline module we will be using is the program **EX_IMAGE.C**. With this module we will describe the minimal requirements for using any d-tree tools as well as illustrate the screen and keyboard I/O facilities.

At this time either print a hardcopy of the file "ex_image.c" and "ex_image.dts" or load them into your text editor when necessary to follow along as we explain the various **d-tree** tools used in it.

The first significant items are the include statements.

#include "DT_DEFIN.H" - ability definitions
#include "DT_GLOBL.H" - global variables

The header files DT_DEFIN.H and DT_GLOBL.H are both required for any modules using **d- tree** functions.

The next section you will see is simply a series of user field definitions. These are ordinary field definitions which will be used in this particular program.

The following section is called a DODA, Data Object Definition Array. Due to the script interface used by **d-tree** ,any program using the tools to manage data must have a DODA. A DODA is simply a table of fields with symbolic names, their associated addresses, the field types and the length of the fields. *(DODAs are also described in the r-tree Reference Manual.)*

The next section is an array of type DTTHRDCD, **d-tree hard coded**. ADAMs can be initialized in static memory by hard coding the specifications into the source of the program. In our example we are hard coding a DODA into the program. The table DTSHRD01, of type DTTHRDCD, is a list of ADAMs that are

hard coded in this source file. This is required for any **d-tree** specification that is hard coded. As we progress into the example this concept of "interpreted" as opposed to "hard coded" definitions will become more clear.

The hard coded table of type DTTHRDCD must contain the following entries for each "Ability" definition that is hard coded.

- The address of the "Ability" definition table. It is possible to have multiple source files with multiple hard coded "Ability" definitions. In order for a hard code table to reconcile the address of the "Ability" definition, this definition must reside in the same source file as the hard coded table (the DTTHRDCD table) or be defined as an extern.

- The reference number (from DT_DEFIN.H) for this "Ability" definition. (ie: DTKDDODA)

- The number of "Ability" definition occurences in this hard coded definition. (ie: number of DODA entries.) This can be coded as: the **sizeof the definition table** *divided by* the **sizeof the typedef** associated with this "Ability" definition.

Since we can have multiple source files, each containing a table of hard coded ADAMs, we must have an additional array, in one of the source files, containing pointers to the hard coded tables. This table must be of type pointer to DTTHRDCD and must be named DTSHRDCD. See below.

```
DTTHRDCD *DTSHRDCD[DTHRDCD]) = {
DTSHRD01,   /* first hard code table */
DTSHRD02,   /* second hard coded table */
};
```

Note: this illustration shows two hard coded tables. If only one is present the second pointer should be replaced with a null pointer such as (DTTHRDCD *)0. Also note, there is a #define in DT_GLOBL.H defining DTHRDCD as 2. This limits the number of separate source files that can contain hard coded **d-tree** "Ability"s to two. To increase this simply change the #define.

As a brief review, we have seen that in any C program using **d- tree** tools you must have:

- The following include files:
  #include "DT_DEFIN.H"
  #include "DT_GLOBL.H"
- user data definitions
- DODA - defining data files (hard coded or interpreted)
- an array of type DTTHRDCD listing all hard coded "Abilitys". (optional)
- an array of type *DTTHRDCD[DTHRDCD] containing pointers to all DTTHRDCD arrays defining hard coded "Ability"s.(optional)

The following section is main(argc, argv). Note that the first function called is the getenv() function. This is a system function which should be supported by your system. If not, replace it with the appropriate function for your particular system. The next items are simply work fields which will be used later in the program.

The **first thing** which **must be done** in the main() portion of any program is to issue a call to **DT_SETTY(1)** to initialize **d-tree**. **DT_SETTY(0)** must be called at the end of each program to deinitialize The function DT_SETTY initializes I/O protocol as well as assigns structure pointers utilized by **d-tree**.

Before we may perform screen I/O we must first inform the program of the terminal and keyboard characteristics. This is accomplished via the "termcap" file. The function DT_KEYBD is passed the "termcap" file name along with the terminal identifier. This function will read the "termcap" file, looking for the desired terminal. Once found, it will initialize this program to that terminal's characteristics. Note the use of the DT_KEYBD function, here we are using the getenv function to get the terminal name.

In this example we are performing screen I/O. Our screens are painted in the file "ex_image.dts". Before they may be used they must first be parsed into allocated blocks of memory. A call to DT_PARSE passing it the screen name ,will result in the parsing of your screen. Syntax errors in the **d-tree** script will be detected. Refer to the error message guide to resolve any errors.

The next statement, the switch statement, illustrates the DT_IMAGE function. In our example we are passing it the value of '1', this is the identifier of the screen, IMAGE(1), in our script. The DT_IMAGE function displays the screen and accepts input from all the fields. The *printf's* following the function call illustrate the recognition of various keystrokes established in the "termcap" file by the terminal definition, as well as data that has been accepted.

Note here that we used a screen number (1). In a **d-tree** script a name may also be used (ie: "master"). The image function calls like DT_IMAGE require an image

number. When a name is used in a script instead of a number, **d-tree** assigns a unique number to that name. This unique number can be accessed within a program with the DT_INAME function.

Let's assume we had painted our screen in "ex_image.dts" as
IMAGE("master")
instead of
IMAGE(1)
We then could replace
switch (kbd=DT_IMAGE(1))
with
switch (kbd=DT_IMAGE(DT_INAME("master")))

As noted above we, call DT_SETTY(0) at the end of every program, followed by the exit call.

This is an example of some basic screen I/O using the **d-tree** tools. We will build on this as we continue to discover the tools.

At this time compile and execute the example program "ex_image.c".

NOTE: While the use of the "termcap" file is still fresh in your mind, it is recommended that you review the TERMCAP section of the Reference Manual at this time.

4.2 Interpreted to Hard Coded Conversion

"Bridging From Interpretive To Hard Coded Entity Definitions"

In the previous session we touched on the concepts of interpretive and hard coded ADAM definitions. A good example of an interpretive definition was the screen that was painted in our **d-tree** script file, read by the parse function and stored in dynamically allocated memory. On the other hand, the DODA definition was hard coded into the C source file and the necessary tables for hard coded defintions were present.

Using the interpretive method to build screens is extremely useful during application development. This provides the flexibility of easily modifying the appearance of a screen, and simply re-parsing, generating instant results. However this does require the additional overhead of parsing.

In this session we will discover how to use the **d-tree** tools to build the bridge from interpretive to hard coded definitions. This capability will provide smaller code as well as increased startup time when the program is executed, because parsing is not necessary. It also allows finished programs to be sent to users without scripts.

By simply adding the function DT_COMPL (*d-tree compile*) to our C source file ("ex_image.c"), **d-tree** will create the C source representation of the script ("ex_image.dts") in hard coded form. This hard coded definition is placed in a header file name passed to the function. This function must be placed after the parse function in your C source code, for it is this parsed definition that it is written to disk.

Place this function in the source file "ex_image.c" just before the final DT_SETTY(0) function call as demonstrated below.

ex_image.c

```
  case DTKBPU:   printf("page up was hit");
                 break;

  case DTKBPD:   printf("page down was hit");
                 break;
  default:
                 break;
 } /* end switch */


/* now lets check some data to see if the input worked */
printf("Number=%d",number);
printf("Name=  %s",Name);

if (DT_COMPL("ex_image.h"))                      Insert function here.
   printf("Could not Write Compile specs\n");

DT_SETTY(0);

exit(0);
}
```

Note that the parameter being passed to DT_COMPL is "ex_image.h". This will be the name of the include file which will contain the parsed script definitions.

Now we must re-compile "ex_image.c" and after a successful compile, execute the program. Nothing obvious will be apparent upon program execution. View the new header file "ex_image.h". This header file was just created by our program with the DT_COMPL function. You will notice the screen and other ADAMs parsed in by DT_PARSE are now residing in this header file as C source code. A table of all the hard coded ADAM's in this source file has also been created. (DTSHRD02). DT_COMPL has also provided the pointer table, for all hard code tables. Note that this pointer table contains an entry for the original table from the source file "ex_image.c" (DTSHRD01), and one for the new table from the current header file (DTSHRD02).

Now that d-tree has created a header file containing the C source code for ADAMs which were previously interpreted via the parse routine, we can now include this header file in our program to replace many previously necessary functions. To illustrate how the source code would now appear including this new header file, view or print the source file "ex_img2.c". Follow along as we compare the differences in this source code and that of our previous "ex_image.c" file. *(hard code vs. interpretive definitions.)*

You will notice the array of pointers to hard coded entity tables has been removed from this source file and placed in the header file. The table in the new header file also now contains an entry for the new table of hard coded definitions within that same header file.
The include statement
#include "ex_image.h
has been added to include the header file containing all the hard coded ADAMs created by DT_COMPL. The file to be included must be the same as that passed to the function DT_COMPL.

Note that the function DT_SETTY(1) must still be called to perform program initialization.

The keyboard definition is now also hard coded. Therefore, we must call DTPKEYST to inform the program that the definition is hard coded. This function is ONLY called when you have a hard coded keyboard definition. *(don't worry, the next sections show hard coded screens with interpreted keyboard)*

In brief, if your keyboard definition is NOT hard coded you must issue a call to DT_KEYBD passing it the "termcap" file name and the terminal name to search for within the "termcap" file. If the keyboard definition IS hard coded you must issue a call to the function DTPKEYST.

Since all of our ADAMs are now hard coded, it is no longer necessary to perform the parsing routine. Thus it has been removed. However, we do need to establish some pointer relationships between the definitions prior to using them. This is accomplished by the set pointers function. (DT_SPTRS). The rest of the program is the same (except we are not calling DT_COMPL).
Within our program you will notice that there are three (3) basic areas of initialization:

- **Program Initialization - DT_SETTY(1)**
  This is required at the beginning of any program using **d- tree** functions. This function sets I/O protocol as well as global variables pertaining to the ADAMs.

- **Keyboard Initialization - DT_KEYBD or DTPKEYST**
  If the definition is to be parsed, the function DT_KEYBD will retrieve the terminal definition from the TERMCAP file.
  If the definition is hard coded, the function DTPKEYST must be called in order for the program to recognize the hard coded definition.

- **Ability Definitions - DT_PARSE or DT_SPTRS**
  If the definition is to be interpreted, DT_PARSE must be called to interpret the d-tree script and create ADAMs in dynamically allocated memory.
  If the ADAMs are hard coded, the function DT_SPTRS must be called to establish pointers to these hard coded definitions.

Note: The following section will illustrate mixing hard coded definitions with those that are interpreted.

THIS PAGE LEFT BLANK INTENTIONALLY

4.3 Combination Interpreted & Hard Coded

In our first example we learned how to define ADAMs by interpreting them at run time. In the next example we learned how to use the DT_COMPL function to hard code these definitions into a header file and include them at compile time. In this session we will discover how to use a mixture of definition methods within the same program. Specifically we will interpret the keyboard definition while leaving all the other ADAMs hard coded as in example two.

First, let's outline the general flow of any **d-tree** parsing function:

- 1. Count the mandatory occurences for each "Ability" definition encountered.
- 2. Dynamically allocate a block of memory of the "Ability"'s type for the number of occurences counted.
- 3. Set the global pointer DTGPOINT[group][ability] to the address of the allocated block of memory.
- 4. Set the global variable DTGNUMBR[group][ability] to the number of ability occurences counted, representing the number of occurences in the allocated memory block.
- 5. Initialize this block of memory with the defintion from the script.

Once the parse is complete the following defines its result:

- An ADAM has been created containing the ability definition encountered by the parse.
- The ADAM's global variables DTGPOINT[group][ability] has been set to the base address of the ADAM
- DTGNUMBR[group][ability] has been set to the number of ability occurences in the ADAM.

NOTE: The keyboard definition can viewed as an ability. The "termcap" file can be considered a **d-tree** script specific to the keyboard ability. The DT_KEYBD function is the parsing function specific to the keyboard ability. The result of calling DT_KEYBD is an ADAM of type DTKKEYBD.

The following outlines the flow of the DT_COMPL function:

1. Loop through all ability reference numbers of the current group.

2. Check the global variable DTGNUMBR[group][ability] for a value.

3. If a value exists it writes the ADAM to disk in C source form.
   If no value exists it skips it.

Because of the numerous types of terminals within the UNIX environment, you may wish to have the keyboard definition parsed at run-time for terminal independence while the remaining definitions can be hard coded into the C source code. This requires restricting the function DT_COMPL from generating the C source code for the keyboard. This is done by simply setting its ADAM's global number of elements to 0.

DTGNUMBR[DTGCURGP][DTKKEYBD]=0

The two global variables, DTGPOINT[group][ability], a double subscripted array of pointers, and DTGNUMBR[group][ability], a double subscripted array of integers, both reside in the header file DT_GLOBL.H. The variable DTGPOINT[group][ability] contains the base address to the ADAM while the value of DTGNUMBER[goup][ability] is the number of occurences of a particular function within a particular group.

The first subscript, [group], contains the group number. This is controlled by the global variable DTGCURGP (*d-tree global current group*).
d-tree initializes this value to 0 and does not alter this value unless the GROUP ability is encountered within a **d-tree** script or is changed by a user program. The group facility allows for more than one ADAM of the same type to reside in memory at one time. Manipulating the group number determines which group (of ADAMs) is currently being used. Reference the GROUP ability in the Reference Manual for specific information concerning how to use the GROUP ability in a d-tree script.

The second subscript, [ability], references the type of ability within the particular block of memory. Within the header file DT_DEFIN.H is a list of #define statements equating every possible ability with an identifier.

Back to the keyboard. By setting its global number of elements to 0 we 'trick' the DT_COMPL function into thinking that there are no ADAM for this keyboard, thereby circumventing the generation of the hard coded definition. We will use the DT_KEYBD function as in example one to interpret the keyboard definiton at runtime.

At this time insert the statement

DTGNUMBR[DTGCURGP][DTKKEYBD] = 0

into the "ex_image.c" source file just before the call to the DT_COMPL function. Compile and run the "ex_image.c" program. Then, view the "ex_image.h" header file once again and note that the keyboard ability definition was not generated as when we first created it. All other ability definitions are created.

```
                              ex_image.c
                break;

  case DTKBPD:  printf("page down was hit");
                break;
  default:
                break;
 } /* end switch */

/* now lets check some data to see if the input worked */
printf("Number=%d",number);
printf("Name=  %s",Name);

DTGNUMBR[DTGCURGP][DTKKEYBD]=0; <-------------- Insert statement here.

if (DT_COMPL("ex_image.h"))
   printf("Could not Write Compile specs\n");

DT_SETTY(0);

exit(0);
}
```

Print or view the file "ex_imag3.c". Compile this program. This example shows the interpreted mode illustrated in out first example together with the hard coded mode illustrated in example two, controlled with the help of #defined. We basically combined both techniques into one. This technique is also used in the source file "dt_score.c" which is the d-tree general mainline used for most standard maintenance programs.

THIS PAGE LEFT BLANK INTENTIONALY

4.4 c-tree INTERFACE

Experienced c-tree developers realize that there are a number of details involved in building and maintaining all the information and parameters required by the the c-tree file handling routines. Below are a number of the issues which must be addressed:

- Parameter Files
- Incremental Structures
- Record Lengths
- Field Offsets for Indexes (Hardware Word Alignment Considerations)
- Record Buffers or 'C' Structures
- Special Data Handling (Packing and Unpacking Variable Length Records)
- Target Value Construction (forming targets for keys).
- Insuring 'Clean' Data (Proper Initialization and Padding)
- Record Locking
- Multi-User Interface (Simultaneous Update of Single Record by Multiple Users)
- Data Portability in varying hardware environments
- Opening, Processing and Closing Data Files

d-tree provides a variety of ways to address these issues. d-tree takes a unique approach in simplifying the task of building the requirements necessary to efficiently use c-tree for database management. The following are a few of the features in d-tree to accomplish these tasks.

- **CATALOG** (dtcatlog) - Capabilites built into the catalog program greatly simplify the following:

  Creation of Parameter Files
  Creation of Incremental Files
  Creation of Record Structures
  Record Length & Field offsets: where issues such as hardware alignments are considered in providing the database record lengths and field offsets for key segments.

- **DODA** (Data Object Definition Array) - In order to support a script interface in d-tree, a table must exist in memory which contains the field symbolic names (used in the script), field addresses, field types and field lengths. This table is known as a DODA. The CATALOG will also generate DODA entries upon request.

- **Record Buffers** - All c-tree calls that 'get' or retrieve data from disk require an address in memory to place the data. This address is the base address of what is termed a 'record buffer'.

d-tree supports two methods of record buffer definition:

- **1) Traditional** - The typical manner that records are used in the C programming language is to build a structure which defines the data record. The address of the structure is what is provided to the c-tree 'get' call. The record is then read into this structure. This method requires hard coding record structures into the source file which must be recompiled if the record definition changes. Using this method, the DODA will have the address of each field hard coded into it. The following diagram illustrates this method. **NOTE**: The catalog provides an easy way of generating C structures as well as DODAs, if the traditional method is used.

```
     struct uco{
TEXT name[30];
TEXT address[30];
}uco[3];


DATOBJ DTSDDODA[] = {
{"name",uco.name,RTSTRING,30},                Hard Coded
{"address",uco.address,RTRSTRING,30},
};
```

- **2) Dynamic** - d-tree can maintain data base records without the need to define the record buffer as a hard coded memory allocation (no C structures are necessary). This is done by supplying a NULL address for the fields in the DODA. This is illustrated in the following diagram.

```
DATOBJ DTSDDODA[] = {
{"name",NULL,RTSTRING, 30},
{"address",NULL,RTSTRING, 30},
};

                 NULL address
```

DT_ALIGN function - Regardless of the method used to define the record buffers, DT_ALIGN must be called at the beginning of your program for initialization. This call should follow two other calls. A call must be made to the following functions first:

- **DT_SETTY** - A call to this function is mandatory in every program and will initialize d-tree pointers.
- **Open Data Files Call** - There are three different methods of opening the data files.
  - - Parameter file method - **OPNISAM** (c-tree)

- - Incremental file open - Combination **INITISAM with OPNIFILS** (c-tree)
- - **DT_IFILS** - Handles all requirements for opening a file. (d- tree)

Once the data files are open, d-tree has established the first and last fields in the DODA for each file. (See first and last parameters in Parameter Files or Incremental Structures.) The DT_ALIGN function will perform the following tasks based upon the record buffering method used.

- Using the **Traditional** record buffering method, the (hard coded C structures) the DT_ALIGN function will verify the address set in the DODA.
- Using the **Dynamic** record buffering method, the DT_ALIGN function detects the NULL address entry in the DODA, dynamically allocates an array of three record buffers for each file then appropriately sets all the field addresses in the DODA. (See Three Buffer Approach below.)

Three Buffer Approach - Multi-user Conflict Checking - As noted earlier the Dynamic method of allocating record buffers automatically creates an array of three record buffers. When record buffers are defined as C structures (the Traditional method) you MUST define an array of records. Three are required as illustrated.

```
     struct uco{
TEXT name[30];
TEXT address[30];
}uco[3]; <--------- 3 Record Buffer Array
```

d-tree utilizes this three record buffer approach to prevent multi-user conflicts. A multi-user conflict would be a situation involving two users attempting to write the same record at the same time. In a typical file maintenance application the following activity involving the three buffers will occur:

- First the record is accessed by a c-tree 'get' call and read into the provided record buffer. For our example we will use record buffer #1. The record, once read into buffer #1, is immediately copied to buffer #2.
-
- Actual modification to the record is performed in buffer #2. When the user commits this to disk, d-tree first acquires a lock upon the record. If a lock cannot be obtained this process fails because another user has the record.

- Once a lock has been acquired, it rereads the original record back into buffer #3 and compares its contents against buffer #1. If no changes were made to this record by another user since the time you originally read the record, buffer #1 and #3 should be identical. If they are different, this is an indication that another user has modified this record and therefore it should not be written to disk, overlaying the other users changes. An error message should be presented to the user stating 'An update cannot be made at this time. Another user has already made updates to this record. Please start your update process again.' The programmer can control the message and the next action. If the comparison is successful, record buffer #2 is written to disk and the record lock is released.

In review, this is a step-by-step explanation of how the three buffer method handles multi-user conflicts.

- Read record from disk into buffer #1
- Copy to buffer #2
- Make changes to buffer #2
- Lock record or Abort process
- Read record from disk into buffer #3
- Compare buffer #3 to buffer #1
- Write buffer #2 to disk or Send Error Message
- Release record lock

Record Lengths and Index Considerations
There are two methods, supported by d-tree, to define record lengths and its index:

- **Traditional** - The Traditional c-tree methods are Parameter files and Incremental Structures. The Parameter File method stores the file definition and related information in a separate file identified by a '.p' extension. The Incremental File method hard codes the same information within your program. (Reference the c-tree manual for more detailed information on these methods of file defintion.)

  However, one problem that still exists for any data file is data portability. For instance, moving data from a 16 bit processor to a 32 bit processor, the first being word aligned while the latter is double word aligned. The record lengths as well as offsets within a structure where fields reside can be different. This problem, experienced when using Traditional methods, can be tedious to the developer who works in numerous environments. d-tree provides a Dynamic solution to this problem.

- **Dynamic** - d-tree provides a means that you can define a method by which data and index file definitions files may be portable to and from differing hardware environments. This is accomplished by not hard coding record lengths, key lengths and key offsets within your incremental structures, since these are the factors which change from environment to environment. By applying special parameter entries to IFIL, IIDX and ISEG definitions, d-tree can provide a portable means of defining your data files.

- **IFIL** - d-tree will calculate the record length under the following conditions: Enter a record length of zero (0) (fixed length file) or the DODA occurrence number of the first variable length field in the record (varaiable length file). In order to recognize the records as being variable length, d-tree references the file mode entry. If the file mode is less than 0, it is assumed that this is a variable length file definition and the value found in the record length is the occurence number within the DODA of the first variable length field. The negative number assigned in the file mode is the valid file mode simply prefixed by a minus ("-") sign. (i.e. file mode = 5 variable length file mode = -5) d-tree will convert this entry to the proper positive representation. d-tree will then determine the record lengths. (the DODA ocurrence number aids in setting the fixed length portion of a variable length file).

- **IIDX** - Index Structures - The key length is the entry which must remain flexible within the IIDX structure. Instead of hard coding the key length enter a -1 to prompt d-tree to calculate the key length.

- **ISEG** - Key Segment Structure - The segment offsets are the entries which must remain flexible. Enter the DODA occurence for the field to be used prefixed with a minus sign ("-"). The negative value indicates to d-tree that this is a DODA occurence and not the actual key segment offset entry. d- tree will strip the minus sign and use that value at run-time to determine the proper physical offset value. **NOTE:** See DTCATLOG.H for a good example of how these techniques are used in the CATALOG program.

c-tree Interface Related Functions

d-tree provides a number of functions to greatly simplify interfacing with c-tree.

DT_IFILS - Opening all the files required in a program. DT_IFILS performs the following tasks:

- If an IFILS ability section has been defined in a d- tree script file, DT_IFILS is used to access the data dictionary and load into memory all the required data file and index file definitions.
- Performs all initialization required for c-tree, counting the required files and performing an INTISAM call.
- Sets any defined portability requirements. (As discussed in Record Lengths and Index Considerations of this chapter.)
- Opens all defined incremental files.
- Optionally rebuilds corrupted files. #define DTRBLIFIL must be must in the header file DT_DEFIN.H.

DT_ADREC - (Add a record to a c-tree file) - d-tree will perform the following tasks in an add condition:

- Pads the records
- Performs any UNIFORMAT logic required
- Determines if records are fixed or variable length
- If variable length, Packs the record
- Acquires a new record position and locks it
- Adds the record
- Frees the new record lock

DT_DLREC - (Delete a record) - d-tree will perform the following tasks in a delete condition:

- Determines if records are fixed or variable length
- Performs appropriate c-tree delete call

DT_RWREC

- (Rewrite a record) - The rewrite operations of d- tree automatically performs the following tasks:

- Performs multi-user interface conflict checking
- Pads the records
- Performs any UNIFORMAT logic required
- Determines if records are fixed or variable length
- If variable length, Packs the record.)
- Performs call to record lock
- Rewrites the record
- Frees the record lock

Data Base "GETS"-

DT_FSREC	Get the first record in a c-tree file.
DT_NXREC	Get the next record in a c-tree file.
DT_PVREC	Get the previous record in a c-tree file.
DT_EQREC	Get a data record with a key value equal to the target value.

The read logic performs the following tasks:

- Determine if fixed or variable length
- Access the record
- Unpack variable length records
- Performs any UNIFORMAT logic required
- Unpads the record

Miscellaneous Data Management Functions

DT_EDRRD	Re-read record; Main function for multi-user conflicts.
DT_DOINT	Initialize d-tree record buffers.
DT_UNPAK	Unpack records.
DT_DOPAD	Pad fixed length fields.
DT_UNPAF	Unpad record structures.
DT_VLOUT	Pack a variable length before writing to disk.
DT_VLINN	Unpack a variable length record after reading from disk.
DT_AVREC	Add variable length records.
UNIFRMAT	Uniformat

THIS PAGE LEFT BLANK INTENTIONALLY

Other d-tree Features

5.1 Print Screens in Xenix/Unix Environment.

d-tree allows a "print screen" capability in environments where a "print screen" is not typically supported. By setting the **#define DT_PRTSCR** in file dt_defin.h, this feature is activated. d-tree is shipped with this #define already set. When the user hits the "print screen" key as defined in the TERMCAP file, the image out function (DT_IMGOT) is called to redisplay the screen. Note: The output functions in d-tree are passed file pointers to where the output is to be directed. Normally DT_IMGOT is called with stdout as the file pointer. When the "print screen" is hit a temporary file is created (using the your runtime lib's **tmpnam** function call). The DT_IMGOT is called with this file pointer "dumping" this screen into this file. The code in "dt_image.c" shown below then prints the file.

dt_image.c

```
#ifdef DOS
  strcpy(prtque,"type  ");
  strcat(prtque,prtfile);
  strcat(prtque," > ");
  strcat(prtque,getenv("PRINTER"));
  system(prtque);
#else
  strcpy(prtque,"lp -d");
  strcat(prtque,getenv("PRINTER"));
  strcat(prtque," -s -c ");
  strcat(prtque,prtfile);
  system(prtque);
#endif
 }
```

When the "print screen" key is hit, d-tree writes the screen to a temorary work file which is printed via this code in file "dt_image.c".

With a little creativity, the DT_IMGOT function can be used to create simply reports, where the output is "painted" as an IMAGE in a d-tree script, and the DT_IMGOT function does the output to the given print file.

5.2 Direct memory video writing/Color Support. (DOS ONLY)

Writing directly to the video memory in order to provide INSTANT screens is controlled by the #define INSTANT in the file dt_defin.h. The logic that actually performs this task is written in assembler and can be found in the directory \INSTANT on the last distribution disk. These modules have been compiled and placed in the library called DTDOSL.LIB in the \INSTANT directory. Program compiled with #define INSTANT must be linked with this library. At this point in time, all color support is done thru direct video memory access, therfore the #define INSTANT must be set for color. Color support also requires a TERMCAP terminal entry that defines the colors. The supplied terminal definition named DOSCOLOR will suite DOS users. We have not used d-tree to control color on ANSI type terminals (wyse, televideo, etc). If these colors can be controlled with esccape sequences, the sequences define in the TERMCAP file should be able to handle it.

TERMCAP

6.0 TERMCAP - Terminal/Keyboard Interface

The **TERMCAP** file is a text file containing definitions of screen and keyboard configuration. These definitions will be used at run time to define the current terminal's characteristics. The format of these definitions has been designed to be straight forward, relating a simple identifier to a decimal sequence(s). These identifiers can be broken down into two different categories, values returned by the keyboard and escape sequences sent to the terminal device. For example, 'CR', Carriage Return, is equated to the decimal representation returned by the keyboard, 13, 'BU', Back Up, is equated to a decimal value of 8, and so on. Closer to the bottom of the file you will see the screen attribute symbols and the string of ESC sequences necessary to produce the attribute. A complete table of delivered symbols may be found in the file dt_keybd.h.

```
TERMCAP
TERMINAL(DOS)
ESC 27 27 CR 13 BU 8 DC 0 83 IC 0 82 DU 0 80 UP 0 72 PD 0 81 PU 0 73
LF 0 75 RT 0 77 HM 0 71 EN 0 79 AD 9
F1 0 59 F2 0 60 F3 0 61 F4 0 62 F5 0 63 F6 0 64 F7 0 65 F8 0 66 F9 0 67 F10 0
F11 3 CTLA 1 HELP 96
LOC 0 27 91 120 59 121 72 CLS 27 91 50 74 EOL 27 91 75
UL 27 91 52 109 RI 27 91 55 109 NA 27 91 48 109 PS 2 HL 27 61
LOTUS 27 91 55 109
FRAME_LTC 201
FRAME_LBC 200        The keyword TERMINAL is required.
FRAME_RTC 187
FRAME_RBC 188
FRAME_HOR 205
FRAME_VIR 186
                        The Terminal Name must immediately follow
                        the TERMINAL keyword in parenthesis.

TERMINAL(wyse50)
ESC 27 27 CR 13 BU 8 DC 127 IC 27 80 DU 10 UP 11 PD 27 74 PU 27 75
LF 8 RT 12 HM 30 EN 27 55 AD 9 F1 27 49 F2 27 50 F3 27 51 F4 27 52 F5 27 53
F6 27 54 F7 27 123 F8 27 56 F9 27 57 F10 27 58 CTLA 1
F11 3
```

The only deviation from the screen attributes found in the TERMCAP file occurs when direct video writes are used.

NOTE: Although **d-tree** is delivered with some example terminal definitions, it is the user's responsibility to construct his own entries. It will be necessary for you to refer to your terminal documentation to obtain the information necessary to create a complete terminal defintion. Although DOS terminal definitions are fairly standard and the default terminal definition will work quite adequately for your needs, there may be some keystrokes you will want to redefine to meet your preferences. ie HELP key, POST key, DEFAULT key, etc. The necessity to create a new terminal definition mainly applies to non-DOS environments.

DTERMCAP-The utility program dtermcap is provided to simplify the insertion of new terminal definition scripts in the TERMCAP file. To insert a new terminal definition in the TERMCAP file start the utility program by entering: **dtermcap**

d-tree will first present an instruction screen defining the valid keystrokes during data entry. These are:

KEY DESCRIPTION

'-' Backup one keystroke definition.
'+' Accept this entry and proceed to the next prompt.
'|' Abort the definition entry operation.

The first data you must enter is a Terminal Name. This name will be used to identify the terminal definition in the TERMCAP file.

```
C:\DT >dtermcap

                    FairCom Termcap Setup
                         d-tree (tm)
                     Copyright (c) 1988


This program will prompt you for a series of keyboard strokes.
As a key is hit, it's decimal representation is echoed to the screen.
Once you have defined the keystroke(s) for a purpose. (Page Up....ect)
press the '+' key to advance to the next definition. The '-'
key is used to back up if you make a mistake. The '|' key is
used a an ABORT key.


Enter Terminal Name...........................
```

dtermcap then proceeds through the keystroke definitions entry by entry prompting you to press the equivalent keystroke on your keyboard.

```
Enter Terminal Name.....................  ..DOS03
Enter Escape Key...                       .....27
Return Key.........                       .....13
Backup Key.........                       .....8
Delete Character...                       .....0 83
Insert Character...                       .....0 82
Down Key...........                       .......0 80
Up Key.............                       .......0 72
Page Down..........                       .......0 81
Page Up.....................................0 73
Left Key....................................0 75
Right Key...................................0 77
Home Key....................................0 71
Post Key....................................0 79
Default Control Key.........................9
Print Screen Key (optional-skip in DOS)........
Help Key............                        0 35
F1 Key..............                     .....0 59
F2 Key..............                     .....0 60
F3 Key..............                     .....0 61
F4 Key..............                     .......0 62
F5 Key..............                     .......0 63
F6 Key..............                     .......0 64
F7 Key......................................0 65
F8 Key......................................0 66
```

Terminal Name
The name used in this entry will be the terminal ID used in the TERMCAP file.

Post Key
This key is used by the CATALOG program to write input to disk. Typically the END key is defined for this purpose.

Default Control Key
This key is used to insert a default value into a data entry field. Typically the TAB key is defined for this purpose.

Help Key
In this example the Alt H key is used to prompt for help.

After completing all the keyboard definitions dtermcap will then begin prompting you for screen attribute definitions. It will probably be necessary for you to have the documentation for the terminal being defined at hand for reference.

```
F9 Key.........................................0 67
F10 Key........................................0 68
F11 Key........................................0 84
F12 Key........................................0 85
F13 Key........................................0 86
F14 Key........................................0 87
F15 Key........................................0 88
F16 Key........................................0 89
F17 Key........................................0 90
F18 Key........................................0 91
F19 Key........................................0 92
F20 Key........................................0 93
Control A Key (optional) ......................1
Position Cursor on Screen......................
      Offset to add to row and column.....
      Locate sequence.....
      (Enter an x for row and y for column)...120 121
Clear Screen Sequence..........................
Clear to End of Line...........................
Normal or No attribute.........................
Underline Sequence.............................
Reverse Image sequence.........................
INPUTRI output attribute.......................
Half Line Feed sequence (optional).............
Frame: Left Top Corner.........................
```

```
Shift + Fkey if
your keyboard only
goes to F10.
```

```
The remaining
terminal definition
entries define
screen I/O. Refer
to your terminal
documentation for
the proper entries.
```

Once all the terminal characteristics have been defined, dtermcap will ask if you wish to test the terminal definition. A 'Y' will test the definition while 'N' will terminate the terminal definition installation session.

Advanced TERMCAP Concepts

The remainder of this section will be given to more detailed information concerning the mechanics of the terminal definitions. Topics such as:

- How these definitions are actually used by d-tree.
- How to add a new terminal characteristic to the TERMCAP file as well as all other necessary files.
- How to include the new terminal characteristic in the dtermcap utility program.

Terminal definitions are treated by d-tree as simply a d-tree script. At run time these terminal definition scripts are parsed by the function DT_KEYBD and stored in a typedef. When calling DT_KEYBD you must pass it two parameters, the TERMCAP file name and the terminal identifier to be parsed. DT_KEYBD will parse the definition script, allocate a block of memory and load the parsed definition into that memory forming an Ability Definition Allocated Memory block (ADAM) of type DTTKEYBD.

dt_typdf.h

```
/**********************************************************************/
/* KEYBD definitions */
#ifdef DTKKEYBD
typedef struct {
COUNT   terminal;           /* terminal id */
COUNT   retcode;            /* what to return if input matches*/
COUNT   frschar;            /* first char of key sequence */
COUNT   noofchar;           /* no of additional char in sequence*/
TEXT    addchar[DT_MXSEQ];  /* additional chacters in seqence */
} DTTKEYBD;

EXTERN COUNT     DTKEYMAP[256]; /* keyboard map array               */

/* note-do not assign (-1) to any screen seq or keyboard key  */
/*       for (-1) is used to detect termination               */
/* remember that all screen sequence numbers must be numbered */
/* sequentually without skipping any numbers between the first */
/* and last number */
/* there are two types of output attributes for a field */
/* screen control seq is one type, and simple output attributes that */
/* do not involve special screen seq is another */
```

Adding A New Keystroke or Screen Attribute Definition

Let's add both a new keystroke definition and a new screen attribute to our terminal definition that is not currently defined in d-tree. In our keyboard definition we will insert a 'HOT KEY' definition and in our screen definition we will add the 'BLINK' attribute.

First we must edit the header file DT_TYPDF.H. Postion yourself at the bottom of this file. Notice that the #define statements for all the screen attributes begin with the prefix DTSC, d-tree Screen Control. e.g. DTSCCLS = d-tree Screen Control CLear Screen. Above these entries are the #define statements for all the keystroke definitions beginning with the prefix DTKB, d-tree KeyBoard.

Within this header file, DT_TYPDF.H, we must enter a new #define statement for each additional definition we wish to insert. We will first add the 'BLINK' screen attribute. Notice the two #define statements:

#define DTSCFRST (-10)

#define DTSCLAST (-23)

NOTE: If the value in parentheses on the second line is not (-23), substitute the correct value whenever the value of DTSCLAST is referenced in this session.

dt_typdf.h

```
#define DTSCFRS    (-10)  /* define first screen sequence number */
#define DTSCLST    (-24)     define last  screen sequence number */

#define DTSCLOC    (-10) /*                                      */
#define DTSCCLS    (-11) /*                                      */
#define DTSCEOL    (-12) /* scr                                  */
#define DTSCNON    (-13) /* non                                  */
#define DTSCUL     (-14) /* scr                                  */
#define DTSCRI     (-15) /* rev                                  */
#define DTSCHL     (-16) /* half line feed */
#define DTSCLOTUS  (-17) /* lotus style */
#define DTSCFMLTC  (-18) /* frame left top corner character */
#define DTSCFMLBC  (-19) /* frame left bottom corner character */
#define DTSCFMRTC  (-20) /* frame right top corner character */
#define DTSCFMRBC  (-21) /* frame right bottom corner character */
#define DTSCFMHOR  (-22) /* frame hoizontal line character */
#define DTSCFMVIR  (-23) /* frame vertical line character */
#define DTSCBLINK  (-24) /* screen blink */

#define DTKBESC   (-30)
#define DTKBCR    (-31) /* Cr-Return on keyboard                 */
```

The DTSCLST, d-tree Screen Control Last, variable must be adjusted.

This is our new screen entry.

These entries, d-tree Screen Control First and
d-tree Screen Control Last
define the boundaries of the screen control sequence numbers. All screen control sequence numbers must be within the boundaries established by these two #define statements and they must be in sequential order.

To add a new screen control attribute the DTSCLAST value must be adjusted to make room. Edit this statement to read:

#define DTSCLAST (-24)

(or add -1 to the negative value of your DTSCLAST)

Insert the statement:

#define DTSCBLINK (-24)   /* screen blinking attribute */

immediately following the last DTSC #define statement.

Next we must insert our new keystroke definition. View the keyboard related #defines. Note that each of them begin with the prefix DTKB. These entries are not restricted by any sequence number boundaries. The only requirements placed on keyboard entries are that the sequence number may not be duplicated, the real value of the sequence number must be greater than that of the real value of the greatest screen control sequence number and all entries must be in sequential order.

Insert the following statement immediately after the last DTKB #define statement incrementing the sequence number:

#define DTKBHOT (-67)   /* Hot Key   */

(The value in parentheses may be different from what you may need to enter.)

dt_typdf.h

```
#define DTKBF6    (-50) /* function key F6           */
#define DTKBF7    (-51) /* function key F7           */
#define DTKBF8    (-52) /* function key F8           */
#define DTKBF9    (-53) /* function key F9           */
#define DTKBF10   (-54) /* function key F10          */
#define DTKBF11   (-55) /* function key F11          */
#define DTKBF12   (-56) /* function key F12           */
#define DTKBF13   (-57) /* function key F13           */
#define DTKBF14   (-58) /* function key F14           */
#define DTKBF15   (-59) /* function key F15           */
#define DTKBF16   (-60) /* function key F16           */
#define DTKBF17   (-61) /* function key F17           */
#define DTKBF18   (-62) /* function key F18           */
#define DTKBF19   (-63) /* function key F19           */
#define DTKBF20   (-64) /* function key F20          */

#define DTKBCTLA (-65) /* control 1                 */
#define DTKBHELP (-66) /* HELP key                  */
#define DTKBHOT  (-67) /* HOT KEY                   */

#endif
/*******************************************************/
```

This is our new keyboard entry.

Save all edits made and exit from editing the DT_TYPDF.H header file.
We must now add our symbol names within the header file DT_KEYBD.H. Within this file is the table that is used to assign the symbol reference that will be used in the TERMCAP file. For clarity in organization only, the keystroke symbols are first, followed by the screen control symbols. Let's insert our new symbols between these two sections. Immediately following the last DTKB entry, insert these two entries:

```
{"HOT"  ,DTKBHOT},      /*Hot Key               */
{"BLINK",DTSCBLINK}, /*Blink               */
```

dt_keybd.h

```
{"F6"  ,DTKBF6},      /* function key F6           */
{"F7"  ,DTKBF7},      /* function key F7           */
{"F8"  ,DTKBF8},      /* function key F8           */
{"F9"  ,DTKBF9},      /* function key F9           */
{"F10"  ,DTKBF10},  /* function key F10          */
{"F11"  ,DTKBF11},    /* function key F11          */
{"F12"  ,DTKBF12},    /* function key F12          */
{"F13"  ,DTKBF13},    /* function key F13          */
{"F14"  ,DTKBF14},    /* function key F14          */
{"F15"  ,DTKBF15},    /* function key F15          */
{"F16"  ,DTKBF16},    /* function key F16          */
{"F17"  ,DTKBF17},    /* function key F17
{"F18"  ,DTKBF18},    /* function key F18
{"F19"  ,DTKBF19},    /* function key F19
{"F20"  ,DTKBF20},    /* function key F20
{"HELP",DTKBHELP},    /* help key                  */
{"CTLA",DTKBCTLA},   /* control A                 */
{"HOT",DTKBHOT},     /* hot key                   */
{"BLINK",DTSCBLINK, /* screen blink              */
{"LOC"  ,DTSCLOC},   /* screen locate             */
{"CLS"  ,DTSCCLS},   /* screen clear screen       */
{"EOL"  ,DTSCEOL},   /* screen clear to the end of line */
```

These two lines contain our two new entries.

We have completed all the necessary steps to make a new keystroke definition and screen ability definition recognizable by the DT_KEYBD parsing function. Therefore, their associated new symbols may now be used in the TERMCAP file.

INSERTING THE DTERMCAP PROMPT

The next step is to include our new keystroke and screen attribute in the TERMCAP installation program, dtermcap. This requires us to edit the header file DTERMCAP.H. Within this file are all the prompts which are used by the dtermcap program.

The order that these prompts appear is the same order the prompts will be printed on the screen. Insert the prompts for both of our new features in whatever order you would like to have them displayed.

```
                         dtermcap.h
{"F15 Key.........................................",DTKBF15},
{"F16 Key.........................................",DTKBF16},
{"F17 Key.........................................",DTKBF17},
{"F18 Key.........................................",DTKBF18},
{"F19 Key.........................................",DTKBF19},
{"F20 Key.........................................",DTKBF20},
{"Control A Key (optional) .......................",DTKBCTLA},
{"HOT Key.........................................",DTKBHOT},
{"Screen Blink....................................",DTSCBLINK},
{"Position Cursor on Screen.......................",DTSCLOC},
{"Clear Screen Sequence...........................",DTSCCLS},
{"Clear to End of Line............................",DTSCEOL},
{"Normal or No attribute..........................",DTSCNON},
{"Underline Sequence..............................",DTSCUL},
{"Reverse Image sequence..........................",DTSCRI},
{"INPUTRI output attribute........................",DTSCLOTUS},
{"Half Line Feed sequence (optional)..............",DTSCHL},
{"Frame: Left Top Corner..........................",DTSCFMLTC},
{"Frame: Left Bottom Corner.......................",DTSCFMLBC},
{"Frame: Right Top Corner.........................",DTSCFMRTC},
{"Frame: Right Bottom Corner......................",DTSCFMRBC},
{"Frame: Holzontal Line...........................",DTSCFMHOR},
```

Here are our two entries for DTERMCAP.H

The final step to making the insertion of our new features complete is to compile the following modules:

DT_INPUT.C
DTERMCAP.C

REVIEW

In review, the addition of a new feature to the TERMCAP file requires the following steps:

- Edit the DT_TYPDF.H header file inserting the appropriate #define statement. If the type of feature being inserted has a last sequence number variable these must be adjusted by -1.
- Insert the new symbol name in the header file DT_KEYBD.H.
- Insert prompts for the new symbols into the header file DTERMCAP.H to be used in the dtermcap installation program.
- Compile DT_INPUT.C and DTERMCAP.C

THIS PAGE LEFT BLANK INTENTIONALY

d-tree Ability Reference Guide.

This section will described the abiliies defined in d-tree.

d-tree ABILITY - The term ability can be used in many contexts. For clarity we will outline what constitutes a d-tree "Ability".

- An "Ability" is an activity to be performed by a program for a particular purpose. (ie: screen I/O)
- An "Ability" is given a reference name. (ie: IMAGE).
- An "Ability" is assigned a reference number in DT_DEFIN.H. (ie: DTKIMAGE)
- An "Ability" definition typedef is defined in DT_TYPDF.H to contain the definition of objects necessary to perform the ability. (ie: DTTIMAGE)
- An "Ability" requires an initialized ADAM containing definitions of its related objects (see section 4 for ADAM definition)
- An "Ability" has a d-tree script interface syntax definition. (ie: IMAGE(master))
- An "Ability" has a parsing function which is used to convert its script definition into an ADAM. (ie: DTPIMAGE)
- An "Ability" has related function calls to perform its objectives which use a pointer to the ability's ADAM for required definition. (ie: DT_IMAGE, DT_IMGOT, DT_IMGIN).

This focuses on each ability, providing specifics relating the the concept listed above.

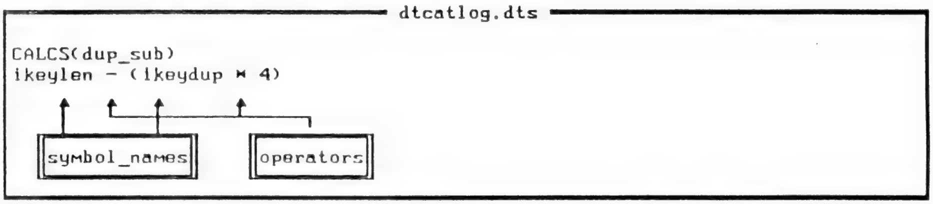
7.1 CALCS - Calulations

DESCRIPTION:

The "CALCS" ability provides the capability to define calculation expressions from within a d-tree script. Typically the CALCS ability will be used in conjunction with symbolic names from the DODA.

SYNTAX:

CALCS(reference_name)
symbol_name1 operator symbol_name2 ...



reference_name- The reference_name must be a unique identifier for each CALCS definition.

symbol_name- The symbol_name must be a field defined within a DODA or a literal value.

operator- The operator must be a valid mathematical operation. The supported symbols are:

+	Add
-	Subtract
*	Multiply
/	Divide

(Note To Advanced Users: The evaluate function (DT_EVALU.C) may be expanded to support additional operators.)

RELATED FUNCTIONS:

- DTPCALCS- Parsing function which initializes the DTTCALCS typedef.
- DT_CALCS- Performs the calculation returning a result either in the form of a long integer or double float.
- DT_CALMP - Performs field mapping with a calculation.
- DT_EVALU - Evaluates a postfix expression.
- DT_PSTFX- Converts an expression in infix form to postfix.

TYPEDEF:
DTTCALCS

```
dt_typdf.h
/****************************************************************************/
/* CALCS definitions */
#ifdef DTKCALCS
typedef struct {
COUNT num;          /* calculation number */
TEXT *string;       /* calculation string */
} DTTCALCS;

#endif
/****************************************************************************/
```

EXAMPLE:
An interest calculation may appear within a d-tree script as follows:

```
CALCS(int_calc)
(custbal * int_rate)/12
```

The following source file example illustrates how the calculation in the d-tree script may be referenced:

```
myfunction()
{
LONG     iresult;
DOUBLE dresult;
COUNT  calcno;
calcno=DT_INAME("int_calc");
DT_CALCS(calcno,&iresult,&dresult);
if(iresult)
    printf("The answer is %ld",iresult);
else
    printf("The answer is %lf",dresult);
return(0);
}
```

INTERNAL d-tree REFERENCE - DTKCALCS

7.2 CONST - Constants

DESCRIPTION:

The "CONST" ability provides the means to define output attributes for constant fields within an image. A 'constant field' is any character or group of characters found on the screen image which is not an input field. This includes field identifiers, headers, titles, special system symbols (prefixed with an '@' sign) and frames (denoted by '+' signs). One or more words that reside on the screen delimited by a single space are grouped together as one constant. Constant values are delimited by a white space or two or more blank spaces.
"CONST" is used in conjuction with "IMAGE". The "reference_name" in parentheses following the keyword "CONST" must be the same as that of the corresponding "IMAGE".
Constants are identified by an associated sequential numbering method. All constant values are counted in sequence beginning at the top left, progressing left to right, top to bottom. (Special note: When determining the sequence number for constants, do not count FRAME TITLES.) Frame Attributes are assigned to the top left corner constant (top left + sign).

SYNTAX:

CONST(reference_name)
value output_attribute

```
CONST(master)
   2 RI
   4 EOL UL RED
   5 LMAG
```

reference_name: The "reference_name" must be identical to that of the corresponding "IMAGE" section.

value: The "value" is the coinciding sequential constant number.

output_attribute: The "output_attribute" is the output attribute to be applied. The following is a list of available output attributes:

RI	Reverse Image	WHITE	white
UL	Underline	GREY	grey
EOL	Clear to End of Line before displaying constant	LBLUE	light blue
		LGREEN	light green
BLACK	black	CYAN	cyan
BLUE	blue	PINK	pink(light red)
GREEN	green	LMAG	light magenta
CYAN	cyan	YELLOW	yellow
RED	red	BWHITE	bright white
MAG	magenta	BROWN	brown

RELATED FUNCTIONS

- DTPCONST - parsing function which initializes the DTTCONST typedef.
- DT_CONST - Display a constant (constant out).

TYPEDEF:

DTTCONST

```
================================ dt_typdf.h ================================
/**********************************************************************/
/* CONST definitions */
#ifdef DTKCONST
typedef struct {
TEXT *string;                  /* pointer to constant text      */
COUNT   len:                   /* length displayed on screen    */
COUNT   outatr[DT_MXOAT]:      /* output attributes             */
COUNT   col:                   /* column number for display     */
COUNT   row:                   /* row number for display        */
} DTTCONST:

#endif
/**********************************************************************/
```

EXAMPLE:

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE   #1                   #2→ FairCom                  #3 → @TIME

   #5                              @CWD ← #4

+Customer Master

#6 Customer Number:  ______
#7 Customer Name:    ______________________________________
#8 Customer Status:  ______
#9 Customer Balance: _______.__

                                                    #10 ——→ +
FIELD(master)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  I/O Mask*/
   custnum      NUMERIC              RI                 1
   custnam      ALLWORDCAPS          NONE               2
   custstatus   ALLCAPS              NONE               3
   custbalance  NUMERIC

CONST(master)
   2  UL
   4  BLUE WHITE
```

The first constant will be underlined. The next will have a forgound of blue with background white

INTERNAL d-tree REFERENCE - DTKCONST

7.3 DEFAULTS - Default field values

DESCRIPTION:

The "DEFAULTS" ability provides various methods of initializing DODA fields with data.

SYNTAX:

DEFAULTS(reference_name)
  symbol_name   type_option   default_value

```
DEFAULTS(master)
/* Symbol Name     Type of defaults     Defaults value */
   custaddress     DFALT_KEY            Unknown
   custstatus      INIT                 OPEN
   activedate      INIT                 SYSDATE
   activetime      INIT                 SYSTIME
   custzip         DUP_DFALT
   custstate       DUP_INIT
```

reference_name- The "reference_name" must be a unique identifier for this DEFAULT definition section.

symbol_name- The "symbol_name" must contain a Symbol Name defined in the corresponding DODA.

type_option - The "type_option" determines how the default entry will be accomplished. The following are valid "type_option" entries:

- **DFALT_KEY** - The default value will be placed in this field when the user presses the Default_Key as defined in TERMCAP. The Default_Key is assigned to the TAB key in the provided TERMCAP. See TERMCAP section to assign a different key to be used as the Default_Key.
- **INIT** - The default value will be placed in the field upon a call to the function DT_DFINI(). The purpose of this function is to perform all initialization type defaults defined in a given DEFAULT definition section. For example- The following steps illustrate the use of the DT_DFINI function when adding a new record to a file:
  1. A record buffer is initialized - DT_DOINT()
  2. Record buffer is initialized with default data values - DT_DFINIT()
  3. Screen image is displayed

  (See the DT_DOINT and DT_DFINI function descriptions.)
- **DUP_DFALT** - The value of this field in the previous record will automatically be duplicated to the current record when the user presses the default key as defined in the TERMCAP file.
- **DUP_INIT** - The value of this field in the previous record will automatically be duplicated to the current record upon execution of the DT_DFINI() function, commonly at initial screen image presentation.

default_value - The "default_value" must contain the data used to initialize the corresponding field. The maximum character length is determined by the length of the destination field. This may be numeric data, a string of characters or one of d-tree's special default values. These special default values are as follows:

- **SYSDATE -** If the entry "SYSDATE" is found, the system date will be placed in the corresponding field.
- **SYSTIME -** If the entry "SYSTIME" is found, the system time will be placed in the corresponding field.

RELATED FUNCTIONS

In this section we will discuss in more detail how the associated functions relate to the various types of defaulting methods.

DT_DFINI

We will begin with the most straight forward variation of the DEFAULT ability, the initialization (INIT) type of default. This form of default entry is performed by the function DT_DFINI. The parameter used by this function is a default number (dfaltno). The DT_DFINI function will initialize all fields in a specific default definition section that are of types INIT or DUP_INIT. Let's assume the following default ability definition for a d-tree script.

```
DEFAULTS(sample1)
 symbol_name          type_option        default_value
      chkflag             INIT                 0
      errflag             INIT                 0
      status              INIT               OPEN
      date              DUP_INIT
```

In general, most d-tree functions reference ability definitions with an associated ability **reference number**. A string of characters cannot be used directly in a function call. For the users convenience, d-tree scripts allow character strings to identify a particular ability definition section. The parsing routine will assign a reference number to each ability definition. To retrieve this reference number, call the function **DT_INAME** passing it the reference_name used in the d-tree script. The following example illustrates how this may appear within your C source file.

```
myfunction()
{
COUNT dfaltno;
dfaltno=DT_INAME("sample1");
DT_DFINI(dfaltno);
}
```

The final result of the **DT_DFINI** function call will be the initialization of the addresses of the symbol_names to the values specified by the default_value for this definition section. The only stipulation placed upon entries into the **symbol_name** field is that they must be defined in a DODA. In this example we are concentrating on the initialization type of default, so we have used both the INIT and DUP_INIT keywords for the **type_option.** The **default_value** should contain the data to be used to initialize the address of the symbol_name. (Note: By using the DUP_INIT type, the contents of the date field will be duplicated from the previous record.)

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE = FairCom                          @TIME

                          @CWD

+Customer Master

  Customer Number: _____
  Customer Name:   ______________________________________
  Customer Address: _________________________________________
  Customer Status: ______
  Customer Balance: _______.__
  Active Date:     ________

                                                             +

FIELD(master)
/* Symbol Name   Input Attribute  Output Attribute  Input Order  */
   custnum       NUMERIC               RI                1
   custnam       ALLWORDCAPS           NONE              2
   custaddress   SCROLL                NONE              3
   custstatus    ALLCAPS               NONE              4
   custbal       PROTECT               NONE              5
   activedate    NONE                  NONE              6

DEFAULTS(master)
/* Symbol Name   Type of defaults   Defaults value */
   custaddress   DFALT_KEY          Unknown
   custstatus    INIT               OPEN
   activedate    INIT               SYSDATE
```

In this example when DT_DFINI is called in the program the customer status field will be set to "OPEN" and the "Active Date" will be set to the system date.

DT_DFIMG - The next form of the INIT default type requires the use of the IMAGE and FIELD abilities. This variation of the INIT default uses the default image function, DT_DFIMG. This function is passed an image number. DT_DFIMG will first search for all fields associated with an image for which an initialization type default (i.e. INIT or DUP_INIT) has been defined. If any are found, they are executed.

The following is an example source file:

```
myfunction()
{
COUNT imageno;
imageno = DT_INAME("master");
DT_DFIMG(imageno);
}
```

DT_DFALT

The DT_DFALT function will perform default entry on one specific field. This function must be passed the field pointer and the type of default to be performed. It will then execute that type of default for that particular field. The DT_IMGIN (image in) function applies this function to handle field defaults. Upon detection of the default key (defined in the TERMCAP), the DT_IMGIN function calls DT_DFALT passing it the current field with the DFALT_KEY type. DT_DFALT will check the default definitions for this field, looking for this type. If one is found the default value will be placed at the field address. Due to the fact that the DT_IMGIN function addresses these situations, most users will not need to be concerned with this level of control. For users who may desire this low level control information, the following illustrates how this entry would appear in both the d-tree script and your source file.

d-tree script:

```
DEFAULTS(master)
/* Symbol Name        Type of defaults      Defaults value */
   custaddress         DFALT_KEY             Unknown
    status             DFALT_KEY              OPEN
```

your source file:

```
myfunction()
{
DTTFIELD *fptr;
COUNT type;
COUNT tlcol;
COUNT tlrow;
fptr = DT_FLDNM("status")
DT_DFALT(fptr,DTDFKEYHIT,tlcol,tlrow);
}
```

The **DT_FLDNM** function, given a field symbolic name, will return a pointer to that field.
The parameter DTDFKEYHIT contains the numeric value representing the appropriate default type. The #define statements establishing these values are defined in "dt_typdf.h" shown here:

```
dt_typdf.h
/* default types */
#define DTDFTAB       1  /* default when auto dup key is hit */
#define DTDFINIT      2  /* default at initlization time */
#define DTDFDUPTAB    3  /* auto dup when auto dup key is hit */
#define DTDFDUPINIT 4  /* auto dup at initlization time */
```

RELATED TYPEDEF:
DTTDFALT

```
dt_typdf.h
/******************************************************************/
/* DFALT definitions */
#ifdef DTKDFALT
typedef struct {
COUNT    num;          /* default number */
COUNT    dftyp;        /* type of default */
TEXT *dftxt;           /* pointer to default text     */
} DTTDFALT;

#endif
/******************************************************************/
```

INTERNAL d-tree REFERENCE - DTKDFALT

THIS PAGE LEFT BLANK INTENTIONALLY

7.4 EDITS - Edit a Field

DESCRIPTION:

The "EDITS" ability provides the means to define specific edits on particular fields along with their associated error messages and other required information.

SYNTAX:

EDIT(reference_name)
error_message symbol_name edit_type edit_information

```
EDITS(example)
/* Error_Message           Symbol      Edit_Type  Edit_Information                   */
Invalid Field Type         cd_fldtyp   VALIDATE   dt_catvd_idx typmap modescan FT
'Y' or 'N' only allowed    cd_resp     TABLE      Y y N n
Field Must Be Entered      td_fil      MANDATORY
Entry Already Exists       td_version  DUPKEY     dt_tdnam_idx
Invalid date  MMDDYY       ddate       DATE_MMDDYY
Invalid date  MMYY         tdate       DATE_MMYY
Filed Must Be Filled       td_code     MAND_FILL
No Entry in Other SFL      ord_itm     SFLSAME    SFLNAME=sflnam SFLFIELD=sflfld
Entry found in other SFL   cd_fldnam   SFLNOTSAME SFLNAME=sflnam SFLFIELD=sflfld
SFL total does not match   ord_amt     SFLHASH    SFLNAME=sflnam SFLFIELD=sflfld
```

reference_name

The "reference_name" must match that of the associated IMAGE and FIELD sections.

Error Message

The "error_message" contains the text to be displayed on the message default line (defined in DT_TYPDF.H by DT_MSGLN) when the specified "edit_type" is violated. This message is delimited by the symbolic name of the field to be edited. Thus, **symbolic names are not allowed** within the edit message.

Symbol Name

The "symbol_name" represents the field to be edited. This field must be defined in the corresponding "FIELD" section.

Edit Type

The "edit_type" contains the valid keyword associated with an edit operation to be performed. If data is entered which conflicts with this edit, the "error_message" is displayed.
All edit types are described in detail below.

Edit Information

The "edit_information" contains additional resources required by specific edit types. Note that not all edit types require this entry.

EDIT TYPES:

- **MANDATORY** - This field must contain valid data.
- **MAND_FILL** - Data must completely fill the field.
- **DATE_MMDDYY** - The data entered must be in the date format of MMDDYY.
- **DATE_MMYY** - The data entered must be in the date format of MMYY.
- **DATE_MMDD** - The data entered must be in the date format of MMDD.
- **TABLE** - The data entered will be compared against a table of data. The table of data is entered immediately following the keyword "TABLE".

TABLE Example:

```
EDITS(custin)
/*Error Message                                      Symbol Name Edit Type  Edit Information  */
Only valid entry is "OPEN" , "CLOSED" or "HOLD"  status     TABLE   OPEN CLOSE HOLD
```

- **DUPKEY** - The "DUPKEY" edit provides a means for preventing duplicate entries in a file where unique keys are defined. Utilizing the provided **index symbolic name**, a target is formed based upon the index segment definition (typically this field will be part of that segment definition). Using this target, the provided index is accessed to determine if an entry of the same value exists.

DUPKEY Example:

```
EDITS(addcust)
/*Error Message          Symbol Name   Edit Type   Edit Information  */
That key already exists    cust_num     DUPKEY    custidx
```

- **VALIDATE** - The validate edit provides the following capabilities:
  1) Insures there is a corresponding entry associated with this field in another file. This field is used as the "target" to access the associated file via the provided index. This index must be defined as unique.
  2) If an associated record is found it allows information from that record to be copied (mapped) into other fields. See MAP ability description.
  3) Because there must be an exact match between this field and an entry in the other file, it provides a means by which the user can perform a look-up (SCAN) into the other file in order to select a valid entry.

VALIDATE Example:

```
EDITS(custin)
/*Error Message           Symbol Name    Edit Type    Edit Information                 */
Invalid Customer Number cust_num      VALIDATE   idxfile codemap codescann prefix
```

- **"idxfile"** - The "idxfile" is the index file to search in validating the entry.
- **"codemap"** - The "codemap" is the reference name for the MAP ability which performs the copying of information if a match is found. See "MAP" ability description.
- **"codescann"** - The "codescann" is the reference name for the SCAN ability which performs the look-up. This allows the user to enter the "lookup character (?)" in the field. See "SCAN" ability description .
- **"prefix"** - Typically the validate keyword will create a target in the same manner as the DUPKEY does. The "prefix" provides a means for this target to be prefixed with a literal value.

- SFLHASH - The "SFLHASH" option will generate a hash total by adding the values in a specified subfile's (SFLNAME) field (SFLFIELD) and verify that hash total against the value in the field being edited. This edit is typically performed when all edits are being performed, prior to posting. This overall edit is performed by the function DT_EDITS. **Note:** DT_EDITS can be called in 1 of 2 modes, edit 1 field or edit all fields. Edits on all fields are typically performed just prior to a post. See DT_EDITS for further information.

SFLHASH Example:

```
EDITS(trans)
/*Error Message     Symbolic Name     Edit Type    Edit Information   */
Invalid Hash Total   tot_fld        SFLHASH   SFLNAME=master SFLFIELD=amount
```

- SFLSAME - The "SFLSAME" option will verify that there is an occurrence of the given field (SFLFIELD) in the specified subfile (SFLNAME) containing the same value. The following example illustrates how the d-tree CATALOG program uses this edit to verify that the field name used in the creation of an index does in fact exist in the "master" subfile.

SFLSAME Example:

```
EDITS(segs)
/*Error Message    Symbol Name  Edit Type    Edit Information */
Invalid Field Name  sd_col  SFLSAME SFLNAME=master SFLFIELD=cd_fldnam
```

- **SFLNOTSAME** - The "SFLNOTSAME" option will verify that there is NOT a field (SFLFIELD) in the specified subfile (SFLNAME) containing the same value. This edit is logically opposite to the previous "SFLSAME" edit type. The following example illustrates how the d-tree CATALOG program uses this edit to insure that an index symbolic name is not the same as a field name.

SFLNOTSAME Example:

```
EDITS(trans)
/*Error Message              Symbol Name   Edit Type     Edit Information           */
Field Name Already Exists cd_fldam  SFLNOTSAME  SFLNAME=master SFLFIELD=cd_fldnam
```

RELATED FUNCTIONS:

- DTPEDIT- Parsing function which initializes the typedef DTTEDITS.
- DT_EDATE- Edit routine - DATES.
- DT_EDITS- Primary field edit function.
- DT_EDSFL- Edit a subfile.
- DT_EDUPK- Edit routine for duplicate keys edit. (DUPKEY)
- DT_EFILL- Edit routine for Mandatory fill edit. (MAND_FILL)
- DT_EMAND- Edit routine for Mandatory field. (MANDATORY)
- DT_ETABL- Edit routine for Table Edit. (TABLE)
- DT_EVALD- Edit routine - Validate with another file. (VALIDATE)

TYPEDEF:

DTTEDITS

```
dt_typdf.h
/*********************************************************************/
/* EDITS definitions */
#ifdef DTKEDITS
typedef struct {
COUNT   edttyp;       /* type of edit */
TEXT *edttxt:         /* pointer to edit message text     */
TEXT *addtxt:         /* additional text */
} DTTEDITS:

#endif
/*********************************************************************/
```

EXAMPLE:

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                          FairCom                          @TIME


                             @CWD

+Customer Master


  Customer Number:   ______
  Customer Name:     _______________________________________
  Customer Address:  __________________________________________
  Customer Status:   _____________
  Customer Balance:  _______.__

                                                                  +
```

```
FIELD(master)
/* Symbol Name  Input Attribute  Output Attribute  Input Order */
   custnum        NUMERIC            RI                  1
   custnam        ALLWORDCAPS        NONE                2
   custaddress    SCROLL             NONE                3
   custstatus     ALLCAPS            NONE                4
   custbal        PROTECT            NONE                5

EDITS(master)
/* Error Message                         Symbol Name Edit Type          */
 Must Enter Customer Complete Number      custnum    MAND_FILL
 Customer Has Already Been Entered        custnum    DUPKEY cust_idx
 Status must be 'OPEN' or 'CLOSED'        custstatus TABLE OPEN CLOSED
```

INTERNAL d-tree REFERENCE - DTKEDITS

THIS PAGE LEFT BLANK INTENTIONALLY

7.5 FIELD - Field definitions

DESCRIPTION:

The "FIELD" ability "ties" input fields of an IMAGE to specific DODA entries via their symbolic names. The "FIELD" ability uses the name in parentheses following the "FIELD" keyword (reference_name) to identify which "IMAGE" definition these fields are associated with. Each line thereafter relates to a specific input field defined on the IMAGE. These definition lines offer a number of options dealing with input and output attributes, cursor control, as well as I/O field masking. Field definition lines must be in the order of their appearance on the screen with direction precedence being top to bottom then left to right.

SYNTAX:

FIELD(reference_name)
/* Symbol Name   Input Attribute {Mask}   Output Attribute   Input Order */

```
FIELD(example)
/* Symbol Name   Input Attribute {Mask}      Output Attribute        Input Order */
    name          FRSWORDCAPS                 NONE                       1
    address       SCROLL                      EOL                        2
    code          TABLE_IN   {!!}             TABLE_OUT                  3
    menu_item     NOCHANGE                    INPUTRI                    4
    state         ALLCAPS                     NOLINES BLACK YELLOW       5
    balance       PROTECT                     RI ZERO                    6
    contact       ALLWORDCAPS                 BWHITE BLUE                7
    zipcode       NONE                        UL                         8
    phone         NUMERIC    {(999)999-9999}  INPUTRI ZERO               9
    ssn           NUMERIC    {999-99-9999}    RED                        10
    date          NUMERIC    {XX/XX/XX}       YELLOW                     11
    time          NUMERIC    {99:99:99}       ZERO                       12
```

Reference_name- The "reference_name" must match that of the associated IMAGE section.

Symbol_name- The "Symbol Name" is the symbolic name for the associated field as defined in the DODA.

Input Attributes- The "input_attributes" define individual field characteristics pertaining to input. The current "input_attributes" are:

- NONE- No special input attribute is applied to this field.
- NOCHANGE- The cursor may enter this field but no modifications may be made. (This option is typically used in creating menus. See "MENUS" description.)
- NUMERIC- All input for this field must be numeric. Only 0-9, +, - or . is valid input for this field.
- ALLCAPS- All alpha characters keyed at this field will be forced to upper case.
- PROTECT- The cursor will not enter this field, thus protecting the field's data from modification.
- FRSWORDCAPS- The first letter of the first word keyed in this field will be forced to upper case.

- **ALLWORDCAPS-** The first letter of each word keyed in this field will be forced to upper case. (A word is determined by a character after a space.)
- **SCROLL -** This field will scroll from right to left when the field length as displayed on the screen is shorter than the actual field length as defined in the DODA.
- **TABLE_IN -** The contents of this field will be converted on input to another value based upon the "TABLES" definition. Requires the use of the "TABLES" ability. (See "TABLES" ability.)

Output Attributes-The "output_attributes" define individual field characteristics pertaining to output. The current "output_attributes" are:

- **NONE -** No special output attribute is applied to this field.
- **RI -** This field will be displayed in reverse image.
- **UL -** This field will be underlined.
- **INPUTRI -** This field will be displayed in reverse image when the cursor enters this field for input. (This is the attribute used to create the reverse image bar which progresses through menu selections.)
- **NOLINES -** The underline characters, or input guide identifying the field position and its length, will not be displayed.
- **EOL -** Before the associated field is output that particular row in the screen image will be erased from the beginning of the field to the end of the line.
- **TABLE-OUT -** The contents of this field will be converted on output to another value based upon the "TABLE" definition. Requires the use of the "TABLES" ability. (See "TABLE" ability.)
- **ZERO -** To be applied to numeric fields. If the value in the field is zero, a zero will be displayed. No underline characters, or input guide identifying the field position and its length, will be displayed.
- **COLOR ATTRIBUTES-**

• BLACK	black	• BLUE	blue
• GREEN	green	• CYAN	cyan
• RED	red	• MAG	magenta
• BROWN	brown	• WHITE	white
• GREY	grey	• LBLUE	light blue
• LGREEN	light green	• CYAN	light cyan
• PINK	pink (light red)	• LMAG	light magenta
• YELLOW	yellow	• BWHITE	bright white

Input Order- The "Input Order" entry determines the field sequence traveled by the cursor upon data entry. By altering these numbers the path the cursor travels will change accordingly.

Input Masks- Input masks present the data field in a more meaningful format. The mask must follow a valid input attribute and must be enclosed in braces. Any constant characters (such as -, /, etc.) may be used within the mask intermingled with valid masking characters. Valid masking characters are:

- X or x - Anything within the regular character set is accepted (no special control characters).
- A or a - Alpha only.
- Z or z - Everthing is accepted (special control characters included).
- ! - Convert to upper case.
- 9 - Numeric only.

Example Masks:

{(999)-999-9999}	example phone number.
{99:99:99}	example time.
{!!}	example first two characters are upper case.
{999-XXXX}	example code field.

RELATED FUNCTIONS: There are a number of functions which work with fields. The following are only the primary field-related functions:

- DTPFIELD - Parsing function which initializes the DTTFIELD typedef.
- DT_FLDIN - Input a field.
- DT_FLDLO - Field out - low level
- DT_FLDNM - Validate token as valid field symbolic name in DODA.
- DT_FLDOT - Display Field (Field Out).
- DT_FLDTX - Convert an Ascii Field to Valid Field Type.
- DT_DFALT - Default a specific field value.
- DT_EDITS - Edit a field.
- DT_HELPP - Provide help for a field.

TYPEDEF:
DTTFIELD

dt_typdf.h

```
/****************************************************************************/
/* FIELD definitions */
#ifdef DTKFIELD
typedef struct {
DATOBJ  *fdoda;               /* pointer to doda            */
COUNT   fdodano;              /* doda number                */
COUNT   len;                  /* length displayed on screen */
COUNT   inpatr;               /* input attribute            */
COUNT   outatr[DT_MXOAT];     /* output attribute           */
COUNT   col;                  /* column number for display  */
COUNT   row;                  /* row number for display     */
COUNT   dec;                  /* decimal positions          */
COUNT   hooks;                /* hook bit mask              */
TEXT    *innmask;             /* input  mask                */
TEXT    *outmask;             /* output mask                */
#ifdef DTOLDHOOK
DT_FPTR funcptr;              /* special function           */
#endif
} DTTFIELD;

#endif
/****************************************************************************/
```

EXAMPLE:

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                          FairCom                          @TIME
                           Customer Master

                              @CWD

+
  Customer Number:  ______
  Customer Name:    ________________________________________
  Customer Address: ___________________________________________
  Customer Status:  ______
  Customer Balance: _______.__

                                                             +

FIELD(master)
/* Symbol Name   Input Attribute {Mask}   Output Attribute        Input Order */
   custnum          NUMERIC                   RI                     1
   custnam          ALLWORDCAPS               NONE                   2
   custaddress      SCROLL                    NONE                   3
   custstatus       ALLCAPS                   NONE                   4
   custbal          PROTECT                   NONE                   5
```

INTERNAL d-tree REFERENCE - DTKFIELD

7.6 GROUP - Group Abilities

DESCRIPTION:

The "GROUP" ability provided a means to "group" ability definitions. An ability definition in its parsed "memory format" is known as an ADAM. *(see section 4 for ADAM discussion)*. Once multiple ADAMS are grouped, they can be written from memory to disk, formulating an "**ability dictionary**". An ability dictionary is a c-tree variable length file that can contain multiple groups. A group is a related set of ability definitions (ADAMS). The concept of a disk file containing "memory-ready" definitions provides a powerful alternative to traditional programming techniques. Data indepedent logic flow can now be written, where the abilities (screens, edits, fields) are no longer hard coded into the program, but "**swapped in and out of memory**" from the "ability dictionary".

GROUPS provides an effective approach to:

- 1) relate ability definitions.
- 2) control swapping definitions ("groups") in and out of memory.
- 3) control **memory utilization** which is especially useful with large d-tree scripts.

SYNTAX:

GROUP(reference name) {FILENAME=filename}

```
GROUP(groupname) {FILE_NAME="ability.dlc"}
```

"**reference name**" must be present to uniquely identify this particular set (GROUP) of parsed ability definitions. There are no prerequisites or dependencies placed upon this entry.

"{**FILENAME=filename**}" - The "**FILENAME**" keyword is optional. If present , the "**filename**" entry is used to identify the name of the c-tree variable length file to be used as the "ability dictionary". If no filename is specified, a default file, named DTGROUP.SWP, will be used.

Let's discuss further how this feature actually works. As explained in section 4, parsing a d-tree script results in an allocated block of memory that is initialized with the script definition for each ability: an ADAM. Logically, the larger the d-tree script, the more memory required to hold all the resulting ADAMs. The GROUP ability provides a way to maintain the current parsed ability definitions so that memory can be "freed" to provide additional space for other ADAMs. This is accomplished by placing "logically related abilities" together in the d-tree script, followed by the GROUP keyword. When the parsing function encounters the GROUP ability section, it will copy all ADAMs currently in memory (in their binary form) to the the c-tree variable length file. The memory previously occupied by those ADAMs is then freed. The parsing process will then continue with memory utitilization reset back to where it was when the parsing process started (as if you where starting to parse a new script).

Remember: when the group keyword is encountered, the definition is saved to disk and memory is freed. Any definition that has been written to disk that is needed by the program must be "swapped-in" before it is used. This is done with the group in function: DT_GPINN. If the last set of abilities in a d-tree script have been "grouped" (the last keyword found in the script is the GROUP keyword), then after the parse is complete, there is nothing in memory that reflects any of the definitions found in the script. All definitions now reside in the "ability dictionary(ies)". The group out function (DT_GPOUT) is the function called by the parsing routine to write the definitions to disk.

RELATED FUNCTIONS:

- DTPGROUP - Parsing function.
- DT_GPINN - Read a GROUP in from disk.
- DT_GPOUT - Write a GROUP to disk.

TYPEDEF: DTTGROUP

```
dt_typdf.h
/******************************************************************************/
/* GROUP definitions */
#ifdef DTKGROUP
typedef struct {
COUNT      num;                /* group number                */
POINTER    recbyt[DTKLAST];    /* record byte position in file */
TEXT       filename[MAXFIL];   /* disk file name for group    */
} DTTGROUP;

#endif
/******************************************************************************/
```

EXAMPLE:

```
IMAGE(file_changed) {LSTFLD_ADVANCE} {BASE_ROW=12} {CLR_BLOCK}
                   +File Definition Has Been Changed

                         Is this a (N)ew version of the file ?
                                              or
                         If so old definition WILL BE LOST.

                                (N)ew or (R)eplace : __

                                                                  +

FIELD(file_changed)
/* Symbol Name   Input Atribute  Output Atribute  Input Order  I/O Mask */
   menu1         ALLCAPS         NONE                1 /* source file name */

EDITS(file_changed)
Must Enter (N) or (R) menu1 MANDATORY
Must Enter (N) or (R) menu1 TABLE N R

GROUP(file_changed) {FILENAME="filchang.sup"}
```

The ADAM created from these ability definitions will be written to this disk file and the memory it occupied will be cleared.

INTERNAL d-tree REFERENCE - DTKGROUP

7.7 HELP - Help Text

DESCRIPTION:

The "HELP" ability provides a means to display help information for specific fields.

SYNTAX:

HELP(reference_name)
USES_SFL(subfile_reference_name)
Help Text or Token fields

```
HELP(master)
USES_SFL(master)
/* Help Text or Token          fields    */
   Place Y or N Here.          prtfield  <-------------- Method #1
   help1                       address1 address2  <---- Method #2
```

"**reference_name**" - The "reference" identifier is used to uniquely identify the HELP section. This section is not dependent upon any other ability section.

There are two methods in which help text may be displayed:

- **Method #1 -** A single line of text displayed on the "default message line". (See #define DT_MSGLN in DT_TYPDF.H.)
- **Method #2 -** Pages of text which may be displayed in a scrollable region on the screen. (subfile)

"**USES_SFL**"- (Method #2 only) - The "USES_SFL" entry is optional. This entry is only used if help text is displayed using a scrollable region of the screen (A subfile). The "USES_SFL" keyword must be followed by a "**(subfile_reference_name)**" -This is the name of the subfile which is loaded with the help text from the help text file. This subfile must be previously defined within a SUBFIL ability section. See "SUBFIL" ability.

"help_text or help_token

Within this entry position you may enter one of two different entries, help_text or a help_token. Enter a single-word identifier to access your help file, method #2, or enter a multiple-word string of help text to be displayed, method #1. The help parsing function (DTPHELP) determines which method of help you are using via the following logic. First, it reads the entry, searching for a valid symbolic field identifier to know when it has reached the end of this entry. If only one word is found by the parsing function when a valid symbolic field identifier is encountered then that single word is

assumed to be a help token. Thus, method #2 is being used. If more than one word is identified within this entry, this string is considered to be help text and method #1 is used.

"help_text" - (Method #1 only) - The "help_text" is the actual text to appear on the screen when help is requested for the associated "field_symbol_name". As noted earlier, a single word is identified as a help token, therefore, help text must be at least two words in length (separated by at least one space). Since the parsing function delimits this entry via a valid symbolic field identifier, symbolic field identifiers are not allowed to be buried within the help text.

"help_token" - (Method #2 only) - The "help_token" is the identifier used to access the associated text from the users help text file. Using this token as the identifier, the associated text is loaded into the defined subfile which is then displayed to the user. This token is the target used for a c-tree FRSSET call.

"field_symbol_name" - (Method #1 and #2) -The "field_symbol_name" is the symbol name(s) as defined in the DODA for which the help information is to be applied. This entry is required for both help methods.

NOTE: When defining help for a field, the help text is assigned to a DODA element. Therefore, help placed on a specific field is available to that field even when positioned on multiple screens.

RELATED FUNCTIONS:

- DTPHELP - Parsing function which initializes DTTHELP typedef.
- DT_HELPP - Help routine.
- DT_BHELP - Build help file index.

TYPEDEF:
DTTHELP

```
dt_typdf.h
/****************************************************************************/
/* HELPP definitions */
#ifdef DTKHELPP
typedef struct {
COUNT   num;            /* help  number */
COUNT   sflno:          /* subfile number */
COUNT   fdodano:        /* doda field number */
TEXT    *string:        /* help text or help token */
} DTTHELPP:

#endif
/****************************************************************************/
```

EXAMPLE:

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                        FairCom                       @TIME

                            @CWD

+Customer Master

  Customer Number:  ______
  Customer Name:    _______________________________________
  Customer Address: __________________________________________
  Customer Status:  ______
  Customer Balance: _______.__

                                                                  +

FIELD(master)
/* Symbol Name   Input Attribute  Output Attribute  Input Order  I/O Mask*/
   custnum         NUMERIC            RI                1
   custnam         ALLWORDCAPS        NONE              2
   custaddress     SCROLL             NONE              3
   custstatus      ALLCAPS            NONE              4
   custbal         PROTECT            NONE              5

IMAGE(helptitle)  {CLR_BLOCK}     +

                                                                  +

IMAGE(help)  {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP}

                                        ____________________________

FIELD(help)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  I/O Mask*/
     help1          NOCHANGE         NOLINES            1

SUBFILE(help)
SFL_IMAGE(help)
SFL_RECORDS(25)
SFL_LINES(5)
SFL_TITLE(helptitle)
SFL_BOUNDARY
/* doda first field       doda last field */
      help1                     help1

HELP(master)
USES_SFL(help)
A customer number must be placed in this field.  custnum
HELP1  custname
HELP2  custstatus
```

HELP TEXT FILE ACCESS

d-tree provides the capability of building an index over a standard text file. This capability is used by the HELPP ability in method #2. By creating a normal ASCII text file containing help text and using unique tokens to identify each section of help text, d-tree provides the means of building an index file over the help text file.

The basic steps in building a help text file and its index are:

- Key the help text into a file. This may be accomplished by using your favorite text editor. Help text for each field should be in separate groups.
- Insert tokens. Each field's group of text should be uniquely identified by a token enclosed in the token delimeters. (The default delimeters are {}).
- Verify filenames. Both the help text filename (help.txt) and the help text index filename (help.idx) are determined within the header file dt_typdf.h. If you elect to use names other than these defaults, the #define statements within this header file should be edited to reflect the proper filenames.
- The program DT_BHELP must be executed. This utility will build the index over the help text file.

The "help file" is simply a standard ASCII text file containing all the help information for each field. The help information for each field should be identified by a unique token enclosed in the defined token delimeters. The default token delimeters are the open and closed braces ({}). The default length of these tokens is 36 characters. The token delimeters along with the maximum token length, help text filename and help text index filename may be altered by editing the header file dt_typdf.h.

```
                              dt_typdf.h
#define      DTHLPFIL   "help.txt"    /* help file data name */
#define      DTHLPIDX   "help.idx"    /* help file index name */
#define      DTHLPTKL   36            /* help token length */
#define      DTHLPLDL   '{'           /* help token left delimiter */
#define      DTHLPRDL   '}'           /* help token right delimiter */
```

Once the help text file has been made, d-tree must build an index file over this help text file for efficient access. This help text index file is constructed by the program dt_bhelp (build help). This utility will first read the entire text file and construct an index key for each token found. Any time the help text file is altered this utility must be rerun to rebuild a new index file for the new version of the help text file.

This key entry assembled in the c-tree index file is defined as follows:

dt_bhelp.c

```
/*

         ----------------------------------------------
        |      c-tree index entry has two parts        |
        |----------------------------------------------|
        |   (anything)        |     (long integer)     |
        |  target portion     |     recbyte portion    |
         -----------^----------------------------^------
                    |                            |
                    ^                            |
                    |                            |
                    ^                            ^
           XXXXX01234                            |
        X..^ .............target.                |
         0......^.......... NULL byte            ^
        1234.....^......... length of text       |
                                                 |
                                                 ^
                   the recbyt portion of the c-tree index is the
                   offset within the text file that the text resides.
*/
```

When a single "help_token" is defined in the d-tree script (method #2), it is used as the target in a c-tree FRSSET call. As shown above, access to the index entry provides not only the offset within the text file where the text starts, but also the length of the text. The DT_HELPP function reads this text and formats the text to the dimensions of the associated subfile. This subfile is then displayed (DT_SFLOT), presenting the help information to the user.

NOTE: You may find this capability of building an index over a standard text file a very useful tool in other appplications. See DT_HELPP.C and DT_BHELP.C.

INTERNAL d-tree REFERENCE -DTKHELPP

THIS PAGE LEFT INTENTIONALLY BLANK

7.8 HOOKS - User Hook into d-tree

DESCRIPTION:

The "HOOKS" ability allows the user to provoke a call to a "**user defined function**" at a specific "hook location" based upon specified conditions.

SYNTAX:

HOOKS(reference_name)

```
/* Hook Symbol Name        Condition                          Function Name  Parameters */
   BEFORE_INPUT            cur_field=amount                     DO_MAP       mapsub
   AFTER_INPUT             cur_field=amount                     DO_MAP       SHOW mapsub
   AFTER_INPUT cur_field=amount AND cur_image=1                 DO_MAP       SHOW mapsub
   AFTER_INPUT cur_field=amount OR  cur_image=master            DO_MAP       SHOW mapsub
   AFTER_INPUT cur_field=amount AND cur_keybd=F1                DO_MAP       SHOW mapsub
```

/* Hook Symbol Name Condition Function Name Parameters */

"**reference_name**" - The "**reference_name**" is required and must be a unique identifier for this HOOKS ability definition section. No prerequisites or dependencies apply to this entry.

"**Hook Symbol Name**"-defines the location in the logic flow where a user-defined function is to be called if the proper conditions are met. The hook ability has been designed to allow the user to add their own hook locations. The steps to add a new hook location are explained in section 9. During the definition of a hook location an associated "Hook Symbol Name" is established. This is the symbol name entered here. The following are the currently defined hook locations:

- BEFORE_INPUT-This hook is located just before a control is given to the user to enter data into the current field.
- AFTER_INPUT - This hook is located just after control returns from the user when entering data into the current field.

"Condition" - The **"Condition"** establishes the qualifying criteria which must be present before the user function is to be called. d-tree has predefined the following conditions:

- **"cur_field="** - the cursor must be positioned within the field reference by the cur_field before the condition is determined as true.
- **"cur_image="** - current image accepting input must be the same as the image defined by cur_image before the condition is determined as true.
- **"cur_keybd="** - the last key pressed on the keyboard must be the same as defined by the cur_keybd before the condition is determined to be true.

(See section 9 to add additional conditions)

The "Condition" entry is allowed to contain boolean (AND/OR) logic. For example: If you only wish to call a specific user function when a specific image (image "master" in our example) is displayed and the user presses a specific key (F1 in this example). This "Condition" entry would appear as follows:

cur_image=master AND cur_keybd=F1

Note: The ADD/OR logic does not support complex expressions. They are simply evaluated from left to right.

"Function Name"- is the name of the user function to be called. This function name must appear as an entry within in the **"user defined function table"**, DTSFUNCT. This table must be defined in every program that calls user defined functions and must be of type DTTFUNCT. This table is used to validate the function name as well retrieve a pointer to be used to call the function. The following is a sample user-defined function table.

```
/* Valid User defined special functions */
DTTFUNCT DTSFUNCT[] = {
{ "DO_MAP",    DT_CALMP },
{ "DT_VERT",   DTCATVRT },
{ "",          DT_NULFP }    /* terminaton Indicator (MANDATORY) */
};
```

"Parameter"- All user-defined functions are passed a pointer of character type (char *) which points the the text defined here in the script. All text following the conditions are passed as a string. The user-define function is responsible for interpreting the information passed in this string.

RELATED FUNCTIONS:

- DTPHOOKS - Parsing function which initializes DTTHOOKS typedef.
- DT_HOOKS - Primary HOOKS function.

TYPEDEF: DTTHOOKS

```
                          dt_typdf.h
/**********************************************************************/
/* HOOKS interface definitions */
#ifdef DTKHOOKS

typedef struct {
COUNT    num:       /* hook number */
COUNT    spot:      /* location (spot) in code where hook occures */
COUNT    if_abil:   /* ability reference number—current ability to check*/
COUNT    if_ocur:   /* ability occurance number—if current ability is this*/
DT_FPTR  funcptr:   /* hook function pointer */
TEXT     *parms:    /* function parameters */
} DTTHOOKS:

#endif
/**********************************************************************/
```

DTTFUNCT - user define function table typdef.

```
/* USER Defined Functions */
typedef struct {
TEXT     *name:        /* function name */
DT_FPTR  funcptr:      /* function pointer to user function    */
} DTTFUNCT:
```

EXAMPLE: One of the most common uses of hooks occurs when you want to accumulate one field from information entered into another field . In the catalog program, we add up the index length as each index segment is entered. Handling this type of approach involves backing out the "old" value of the field, and then adding in the "new". In the following example we define a BEFORE_INPUT hooks to back out the old, and then an AFTER_INPUT to add in the new. The user-defined function DO_MAP uses the MAP and CALCS abilities to perform the data manipulation.

EXAMPLE:

```
IMAGE(index) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP}

   ____________  ____     __    _     _      ___     __ ____ __________

FIELD(index)
/* Symbol Name  Input Attribute  Output Attribute Input Order  */
   id_idx           NONE              NONE         1 /*index name*/
   ikeylen          PROTECT           NONE         2 /* key length*/
   ikeytyp          NONE              NONE         3 /* key type*/
   ikeydup          ALLCAPS TABLE_IN  TABLE_OUT    4 /* duplicate flag*/
   inulkey          ALLCAPS TABLE_IN  TABLE_OUT    5 /* null key flag*/
   iempchr          NONE              NONE         6 /* empty character*/
   id_ifilmod       NONE              NONE         7 /* index file mode*/
   id_ixtdsiz       NONE              NONE         8 /* index file ext size*/
   id_idxfil        NONE              NONE         9 /* index file name*/

IMAGE(segs) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP} {BASE_ROW=14}
                     ____________    ____    ___    __

FIELD(segs)
/* Symbol Name   Input Attribute  Output Attribute  Input Order   */
   sd_col           NONE              NONE            1  /*column*/
   soffset          NONE              NONE            2  /*segment offset*/

   slength          NONE              NONE            3  /*segment length*/
   segmode          NONE              NONE            4  /*segment mode*/

CALCS(segcalc_sub)
ikeylen - slength

MAP(segmap_sub)
/* source field          desination field    Map Type    length */
    segcalc_sub               ikeylen        DO_CALC

CALCS(segcalc_add)
ikeylen + slength

MAP(segmap_add)
/* source field          desination field    Map Type    length */
    segcalc_add               ikeylen        DO_CALC

HOOKS(segs)
/* Hook Symbol Name         Condition            Function Name  Parameters   */
    BEFORE_INPUT            cur_field=slength     DO_MAP        segmap_sub
    AFTER_INPUT             cur_field=slength     DO_MAP        SHOW segmap_add
```

The next example shows how to provoke a user call when processing a certain image if a certain function key is hit:

```
HOOKS(trans)
/* Hook                 Condition                 Function Name  Parameters */
 AFTER_INPUT cur_image=master AND cur_keybd=F11     DT_VERT          1
 AFTER_INPUT cur_image=master AND cur_keybd=F12     DT_VERT          2
```

INTERNAL d-tree REFERENCE - DTKHOOKS

7.9 IFILS - Incremental Files

DESCRIPTION:

The "IFILS" (Incremental Files) ability provides a means to define file(s) and index(es) definitions from a d-tree script. IFILS are used in two ways:

SYNTAX:

IFILS	or	IFILS
DICTIONARY filename		FILE_NAME filename
FILE_NAME filename		KEY_NAME indexname
FILE_VERSION version		KEY_FIELDS field_symbol(s) DUPS_OK
		FIRST_VLEN field_symbol

```
IFILS
DICTIONARY data_dictionary
FILE_NAME  file_name          <----- Data Dictionary Approach
VERSION    file_version
```

```
IFILS
FILE_NAME  data_file_name
KEY_NAME   key_symbol_name    <----- Run-Time Approach
KEY_FIELDS field_symbol(s) DUPS_OK
FIRST_VLEN field_symbol
```

IFILS provides the necessary file and index definition needed to access the data base. The method to use (or to use IFILS at all) depends upon the manner chosen to define the files. Consider the following manners by which files can be defined:

NOT IN d-tree Script - Files and index definitions can be defined in the program either in the form of hard coded incremental files structures or by the use of parameter files. These are the primary methods to define files and index(es) in c-tree, in which case, the IFILS keyword is NOT applicable.

IN d-tree Script-

Data Dictionary Once data file and index definitions have been entered into a data dictionary (Catalog Program), these file definitions can be accessed by programs using the IFILS keyword. Following the IFILS keyword in your script enter the word "DICTIONARY" with an optional file name. (d-tree will default to the catalog's data dictionary file name if no alternative file name is provided). Indicate the file of choice with the "FILE_NAME" entry. The "VERSION" keyword is optional. If provided it will qualify the precise file. If not provided, access to the data dictionary will be done with a LSTSET (see c-tree) retrieving the last version (assumed to be the most current) of the file. This will directs d-tree to the source of the file and index definitions. The call to DT_IFILS in the program will detect this definition, access the data dictionary, and initialize the program with all file and index necessities, including: DODA, IFILS, IIDX, and ISEG structures. **NOTE: Make IFILS the FIRST ability in your script.**

IN d-tree Script-

Run-Time Definitions: As you have seen with the "RUN" program, the d-tree tools have the ability to create file definitions at run-time soley from the d-tree script. Let's talk about how this is done. When an IMAGE keyword is encountered while parsing a d-tree script, a determination is made if there are any file definitions at this time (it checks to see if a DODA already exists). If not, it will build a DODA while parsing, based on the fields found in **this** IMAGE section. Field names default to F001, F002...etc. The DODA provides the fields and the record length to the program, but this is not everthing that is needed to open a file. The IFILS keyword in this case is used to supply the additional information to complete the file and index definitions:

FILE_NAME - provides the data file name on disk to access.

KEY_NAME - provides the index symbolic name.
The first occurence of this entry is considered the primary key while following entries are considered secondary keys (members). The "**symbolic_key_name**" entry will be the symbolic name assigned to the index.

KEY_FIELDS - identifies the field or fields to be used as the key. The first occurance of a "KEY_FIELDS" entry is considered the primary index while subsequent occurences are index members. The "**symbolic_key_field_name(s)**" must be a valid symbolic field name(s) identifying the key segments.
The "**DUPS_OK**" option informs d-tree that duplicate keys are allowed and c-tree duplicate key logic is to be used. This entry is optional and must follow the last field (segment) symbol.

FIRST_VLEN - option identifies that this file is to be defined as a variable length file. The "symbolic_field_name" following the "FIRST_VLEN" option must be the symbolic name of the first variable length field.

The "RUN" program defaults these values when it creates the d-tree script. Using the option as above the user can modify the script to: add additional keys; change key segments; make the file variable length; change the index or data file names on disk; change index symbol names; change field names. NOTE: because the files were already created the first time you ran the "RUN" program, if a change is made, the files must be deleted before the program is run again. This will allow "RUN" to re-create the files with your new definition.

Because the doda is built based on fields found in an IMAGE section if no previous DODA is found, this method only supports one data base file at a time. d-tree only uses this method in the "RUN" program at this time. We do not expect or encourage this approach in your normal development. As with the

"RUN" program, we can see a use for this for "get up quick" or "RUN type" programs. The dynamics involved here are interesting. In one way this "run-time" file definition can be thought of as a VIRTUAL file approach, a concept bound to mature in future releases.

RELATED FUNCTIONS:

- DTPIFILS - Parsing function which initializes the DTTIFILS typedef.
- DT_IFILS - Function to initialize file definitions and open files.

TYPEDEF:
DTTIFILS

```
ctifil.h
#define DAT_EXTENT   ".dat"
#define IDX_EXTENT   ".idx"

typedef struct iseg {
   COUNT soffset, /* segment offset */
      slength, /* segment length */
      segmode; /* segment mode      */
   } ISEG;

typedef struct iidx {
   COUNT ikeylen, /* key length      */
      ikeytyp, /* key type    */
      ikeydup, /* duplicate flag */
      inulkey, /* null ct_key flag  */
      iempchr, /* empty character   */
      inumseg; /* number of segments   */
   ISEG   *seg;        /* segment information  */
   TEXT   *ridxnam;    /* r-tree symbolic name */
   } IIDX;

typedef struct ifil {
   TEXT   *pfilnam;    /* file name (w/o ext)  */
   COUNT dfilno;       /* data file number  */
   UCOUNT   dreclen: /* data record length    */
   UCOUNT   dxtdsiz: /* data file ext size    */
   COUNT dfilmod: /* data file mode */
   COUNT dnumidx: /* number of indices */
   UCOUNT   ixtdsiz: /* index file ext size   */
   COUNT ifilmod: /* index file mode   */
   IIDX   *ix:     /* index information */
   TEXT   *rfstfld:  /* r-tree 1st fld name  */
   TEXT   *rlstfld:  /* r-tree last fld name */
   COUNT tfilno:     /* temporary file number*/
   } IFIL:
```

EXAMPLE:

```
IFILS
DICTIONARY datad.dat
FILE_NAME  myfile.dat
VERSION    1.0


IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                               FairCom                              @TIME
                                Customer Master

                                     @CWD

+

Customer Number:     ______9
Customer Name:       _________________________
Customer Address:    ______________________________
Customer Status:     ______
Customer Balance:    ______.__

                                                          +

FIELD(master)
/* Symbol Name  Input Attribute  Output Attribute  Input Order   */
   F0001          NONE               NONE              1
   F0002          NONE               NONE              2
   F0003          NONE               NONE              3
   F0004          ALLCAPS            NONE              4
   F0005          PROTECT            NONE              5
```

INTERNAL d-tree REFERENCE - DTKIFILS

7.10 IMAGE - Screen Image

DESCRIPTION:

The "IMAGE" ability provides the means to define screens (images) to d-tree. Variable input fields, constant output fields, and special effects such as frames are defined in a WYSIWYG syntax in the d-tree script.

SYNTAX:

IMAGE(reference_name) {option1} {option2} ...

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                         FairCom                          @TIME

                             @CWD

+Customer Master


  Customer Number:  ______
  Customer Name:    ________________________________________
  Customer Address: ___________________________________________
  Customer Status:  ______
  Customer Balance: _______.__

                                                                +
```

A screen is defined in d-tree by means of the IMAGE ability. Enter the keyword IMAGE along with a reference name to start the screen definition.

"(reference_name)" - This entry must be a unique IMAGE section identifier and must match the reference_name of a FIELD section if input fields exist on this screen image.

Optional "IMAGE level" attributes may then be defined by placing the attribute keyword with { } after the IMAGE keyword. These attributes are described below. The first line after the IMAGE keyword is line one of your screen. Paint your screen observing the following points:

- **Constant fields** are keyed just as they are to appear to the user.
- **Variable Fields** are defined by using multiple underscore characters (_) and must be delimited on both sides by a white space (ie: blank, new line, tab, cr). *Note: the (_) is a #define in dt_typdf.h.*
- **Floating point** variables properly display the decimal precision by placing a decimal point at the appropriate location within the (_)'s. Example: ________.__

- The "IMAGE" ability has other special features. By placing these optional entries anywhere within your screen the corresponding system information or feature will be displayed:

  @Date display the **System Date.**
  @Time display the **System Time**.
  @CWD display the **Current Working Directory**.
  (see dt_const.c to add your own special @)

FRAMES:

+1..9 - Frame definition. By placing two plus signs (+) on the screen image you may define the top left and lower right corners of a frame. Up to nine boxes may be defined on a single screen image. If only one is defined, the number one after the plus sign is optional.

Frame Options:

Frame Title- To center a title in the top line of the frame, simply key the title immediately following the top left corner plus sign.

Frame Sides - Which sides of a frame you want to be displayed can also be controlled. If no special control is placed on the frame, all four sides are presented. The keywords {TOP} {BOTTOM} {RIGHT} {LEFT} can be placed directly after the bottom right plus sign (+) to specify the sides to display.

Frame Types - More than one frame type can be defined in the TERMCAP file as long as it only takes a single byte to define the frame character. Terminals that require a multi-byte sequence to display a frame character can only support one frame type. DOS environments typically only need one character, so the terminal definition in the TERMCAP file for DOS terminals have four different frame types. (ie: single bar, double bar, combination bars). To specify the frame type in the IMAGE section enter {FRAME_TYPE=X} where X = a number from 1 to 4 for the desired frame type. See ex_popup.c in section for for example.

"IMAGE LEVEL" OPTIONAL KEYWORDS:

- NO_CLS - The "NO_CLS" option specifies that the screen should not be cleared before displaying the image.
- LSTFLD_ADVANCE - The "LSTFLD_ADVANCE" option directs the program flow to exit the input loop when a "CR" is encountered while the cursor is positioned within the last field on this image.
- FRSFLD_BACKUP - The "FRSFLD_BACKUP" option directs the program flow to exit the input loop when the cursor is positioned in the first input field and the backup key is pressed.
- INPUT_ADVANCE={n fields} or {key} - The "INPUT_ADVANCE" option directs the program flow to exit the input loop when a specified number of fields are input or a specified key is pressed.
- CLR_LINES={n} - The "CLR_LINES" option will clear the specified number of lines from the base of the image being displayed.
- CLR_BLOCK - The "CLR_BLOCK" option will clear only the block of the screen that is used by the image being displayed.
- CLR_EXIT - The "CLR_EXIT" option will clear only the block of the screen that this image uses upon exiting the screen.
- POP_UP - The "POP_UP" option will clear only the block of the screen that this image uses before it is displayed and will then initialize all resources utilized by the DT_UNPOP function. DT_UNPOP will clear the current pop-up image and refresh any previously displayed images.
- BASE_ROW={n} - The "BASE_ROW" option determines the row for the top edge of the image being displayed.
- BASE_COLUMN={n} - The "BASE_COLUMN" option determines the column for the left edge of the image being displayed.
- BACKGROUND = {color} - the "BACKGROUND" option will set the background color (DOS).
- FORGROUND = {color} - The "FOREGROUND" option will set the foreground color (DOS).

  (NOTE: Color keyword definitions are found in the Output Attributes definitions of the FIELD ability.)

RELATED FUNCTIONS:

- DTPIMAGE - parsing function.
- DT_IMAGE - IMAGE OUT then IMAGE IN. Combination DT_IMGOT then DT_IMGIN.
- DT_IMGAL - IMAGE ALL - Display and Input a range of IMAGEs.
- DT_IMGIN - Input an Image (Image In). Series of DT_FLDINs related to the provided IMAGE.
- DT_IMGLG - Redisplay IMAGEs from log.
- DT_IMGMV - Same as DT_IMAGE but allows user to change IMAGE coordinates.
- DT_IMGOT - Display IMAGE (Image Out). Series of DT_FLDOT and DT_CONST related to provided IMAGE.

TYPEDEF:
DTTIMAGE

```
dt_typdf.h
/**********************************************************************/
/* IMAGE definitions */
#ifdef DTKIMAGE
typedef struct {
COUNT    num;          /* image number                                   */
COUNT    cls;          /* clear screen flag                              */
COUNT    lstcr;        /* exit on last field carride return              */
COUNT    fstbu;        /* exit on first field backup                     */
COUNT    inpno;        /* if this many fields have been entered then exit */
COUNT    topcol;       /* Top left corner column for display             */
COUNT    toprow;       /* Top left corner row for display                */
COUNT    basecol;      /* Top left corner column from parse              */
COUNT    baserow;      /* Top left corner row from parse                 */
COUNT    noofvar;      /* number of variable fields                      */
COUNT    noofcon;      /* number of constant fields                      */
COUNT    fstrow;       /* first row                                      */
COUNT    lstrow;       /* last row                                       */
COUNT    lftcol;       /* left most column                               */
COUNT    ritcol;       /* right most column                              */
TEXT     *varptr;      /* first variable relate ptr                      */
TEXT     *conptr;      /* first constant relate ptr                      */
} DTTIMAGE;
```

EXAMPLE:

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                              FairCom                          @TIME

                                  @CWD

+Customer Master


  Customer Number:   ______
  Customer Name:     ________________________________________
  Customer Address:  ___________________________________________
  Customer Status:   ______
  Customer Balance:  _______.__

                                                                  +
```

The following example source file illustrates how the above IMAGE may be referenced and displayed.

```
ex_image.c

imageno=DT_INAME("master");
switch ((kbd=DT_IMAGE(imageno)))  /* project and accept input from image  */
 {
  case DTKBESC:  printf("escape key hit");
                 break;

  case DTKBPU:   printf("page up was hit");
                 break;

  case DTKBPD:   printf("page down was hit");
                 break;
  default:
                 break;
 } /* end switch */
```

INTERNAL d-tree REFERENCE - DTKIMAGE

THIS PAGE LEFT BLANK INTENTIONALLY

7.11 MAP - Field Mapping

DESCRIPTION:

The "MAP" ability provides the means to define "field" mapping to d-tree. By field mapping we mean the ability to copy (map) the contents of one field into another. Data type conversions are supported as well as special map considerations (types).

SYNTAX:

MAP(reference_name)

source field destination field map type length

```
MAP(codemap)
 /* source field   destination field  map type      length */
    code              custstatus       NO_REPLACE      6
    stat              ordstatus        REPLACE
    amt_calc          net_amt          DO_CALC         9
```

- **"reference_name"** - The "reference_name" must be a unique identifier for each MAP section. There are no prerequisites or dependencies upon the MAP ability's reference_name.
- **"source field"** - The "source field" may contain either the symbolic name of the field that contains the data to be copied or a reference name to a "CALCS" ability definition.
- **"destination field"** - The "destination field" is where the data will be copied to.
- **"map type"** - The "map type" will define exactly how the mapping process is to occur. Valid entries for this field are described below.
- **"length"** - The "length" is the number of bytes to be copied. If no length is specified the length of the shorter of the "source" or "destination" fields is used.

MAP TYPES

The following is a description of the valid "map type" entries:

- REPLACE - The contents of the "source field" will replace the contents of the "destination field".
- NO_REPLACE - If data exists in the destination, the map is not executed and the contents of the destination field remains unchanged. (Numeric Fields: Existing data = non-zero)
- DO_CALC- The "DO-CALC" map type maps (copies) the result of a calculation into the destination field. The source field must contain the reference name of the appropriate "CALCS" ability definition.
  Rather than using a symbolic DODA field name in the "source field" entry, a reference name identifying a CALCS definition section is used. After performing this calculation, the result is used as the source data to be used in the MAP. This result is copied into the destination field.

RELATED FUNCTIONS:

- DTPMAPIT - Parsing function.
- DT_MAPIT - MAP routine. MAP one field into another.

TYPEDEF: The MAP ability uses the RELATE typedef DTTRELAT.

```
dt_typdf.h

/* Keyword Relationships */
typedef struct {
UTEXT type;          /* relationship type          */
TEXT *lptr;          /* pointer to left structure  */
COUNT lcnt;          /* left pointer count         */
COUNT ltyp;          /* type of left structure     */
COUNT lsrt;          /* left structure alt sort    */
TEXT *rptr;          /* pointer to right structure */
COUNT rcnt;          /* right pointer count        */
COUNT rtyp;          /* type of right structure    */
COUNT rsrt;          /* right structure alt sort   */
} DTTRELAT;
```

EXAMPLE: As you may recall from the tutorial, the MAP ability is also used by the VALIDATE edit. When a selection is made from a seperate file, the data may be copied back to the data entry field. The following is an example of how the MAP is used in conjunction with the VALIDATE edit. Review the tutorial for further illustration of this feature.

```
IMAGE(master) {LSTFLD_ADVANCE} {FRSFLD_BACKUP}
@DATE                        FairCom                        @TIME


                            @CUD

+Customer Master

  Customer Number:  ______
  Customer Name:    ________________________________________
  Customer Address: ___________________________________________
  Customer Status:  ______
  Customer Balance: _______.__


                                                                      +
FIELD(master)
/* Symbol Name   Input Attribute  Output Attribute  Input Order  I/O Mask*/
   custnum       NUMERIC            RI                1
   custnam       ALLWORDCAPS        NONE              2
   custaddress   SCROLL             NONE              3
   custstatus    ALLCAPS            NONE              4
   custbal       PROTECT            NONE              5

EDITS(master)
 Must Enter Customer Number              custnum   MAND_FILL
 Customer Has Already Been Entered custnum   DUPKEY cust_idx
 Invalid Status     custstatus   VALIDATE   cod_idx codemap codescann prefix

MAP(codemap)
 /* source field          destination field        length */
    code                    custstatus               6
```

The following diagrams show how the map definitions are defined in the DTTRELAT structure.

```
                     MAP FIELD Relationship
        ------------------------------------------------------
        |   ** left side **       |*|   ** right side **     |
        ------------------------------------------------------
        |  |ltyp | lptr  |lcnt|lsrt|*|rtyp |rptr   |rcnt|rsrt|
        ------------------------------------------------------
       /* REPLACE map type     */
       /* NO_REPLACE map type */
        ^...^.......^.......^...^......^.....^.........^...^.. DTRTFMAP flag
            ^.......^.......^...^......^.....^.........^...^.. DTKDDODA
                    ^.......^...^......^.....^.........^...^.. DODA pointer
                            ^...^......^.....^.........^...^.. DODA Number
                                ^......^.....^.........^...^.. map number
                                       ^.....^.........^...^.. DTKDDODA
                                             ^.........^...^.. DODA pointer
                                                       ^...^.. DODA Number
                                                           ^.. map length with
                                                               map type.


      /* CALC map type          */
      /* here we relate CALCS to the RELAT representing
         a specific IMAGE to a FIELD */
        ^...^.......^.......^...^......^.....^.........^...^.. DTRTFMAP flag
            ^.......^.......^...^......^.....^.........^...^.. DTKCALCS
                    ^.......^...^......^.....^.........^...^.. CALCS pointer
                            ^...^......^.....^.........^...^.. CALCS Number
                                ^......^.....^.........^...^.. map number
                                       ^.....^.........^...^.. DTKRELAT
                                             ^.........^...^.. RELAT pointer
                                                       ^...^.. RELAT Number
                                                           ^.. map length with
                                                               map type.

If the type of map is a CALC type then the left side points to the CALCS
definition otherwise it pints to the DODA definition.

If the destination field relates to an IMAGE then the IMAGE to FIELD
RELAT is stored on the right hand side. If the destination field does
not relate to an IMAGE then the right hand side is a DODA entry.

rsrt: We add the map length to the map type value and then
split them back apart in the map function. See map types below.

definition in dt_typdf.h
#define DTRTFMAP 30 /* Relationship contain a field to field map definition */

/* MAP types */
#define DTMAPCAL 11000  /* CALC map type        */
#define DTMAPRPL 12000  /* REPLACE map type     */
#define DTMAPNRP 13000  /* NO_REPLACE map type */
```

INTERNAL d-tree REFERENCE - DTKMAPIT

THIS PAGE LEFT BLANK INTENTIONALLY

7.12 MENU - Menu Support

DESCRIPTION:

The "MENU" ability provides the definition for managing a variety of menu types. Menu presentation and branching is supported by this ability.

SYNTAX;

MENU(reference_name)
 USES_IMAGE(image_reference_name)
/* Call Criteria Type of Call Call Value */

```
MENU(1)
 USES_IMAGE(1)
/*     Call Criteria          Type of Call          Call Value */
         option=1               SYSTEM                dtcatlog
 CURSOR=field option=2          EXECL                 dt_cattd
         option=3               RETURN                3
         option=4               CALL                  demo_4
         option=5               CALLMENU              menu1
```

"**reference_name**" - The "reference_name" must uniquely identify this MENU ability definition section. No prerequisites or dependencies are placed upon the MENU "reference_name".

Note: Before proceeding to define the associated optional keywords, let's first discuss how this ability works. Menus are processed within your program when the related function DT_MENUS is called. DT_MENUS will first display an IMAGE, accept input from the IMAGE and perform the appropriate action ("type of call") based upon the input and/or cursor position specified in the "call criteria" .

"**USES_IMAGE**" - The "USES_IMAGE" keyword must be present and is used to identify the IMAGE to be displayed and accept input. This identification is accomplished via the "**image_reference_name**". The "USES_IMAGE" must be present and must 'tie' to a previously defined "IMAGE" ability.

"**Call Criteria**" - The "Call Criteria" defines what circumstances must exist to initiate execution of its associated call. Within the "call criteria" you may enter two different forms of criteria which will define the circumstances. The first type of criteria is based upon the value of a field (field=value e.g. select=1). The second form of criteria is determined by the field that the cursor was last on (cursor=field e.g. cursor=select). These criteria may be used individually or together in an "and" logic relationship. When used together, both criteria must be true before the appropriate action is taken. Multiple lines of "call criteria" may be used in conjunction with a single menu.

"Type of Call" - When the call criteria is determined to be true, the type of call issued will use the given "call value" to perform "call". The "Type of Call" may be one (1) of four system keywords.

- **EXECL** - The "EXECL" option will call the program defined in "call value" with a "execlp" call, turning program control over to the "called" program and freeing the "caller" from memory.
- **SYSTEM** - The "SYSTEM" option will provoke a "system" call on the program name defined in the "call value", allowing control to return to the primary program.
- **CALL** - The "CALL" option will call the function defined in the "call value". This function must be defined in the "user define function table" (DTSFUNCT).
- **RETURN** - The "RETURN" option will define that the DT_MENUS function is to return the value defined in the "call value".
- **CALLMENU** - The "CALLMENU" option will execute another MENU identified by the reference name defined in the "call value". The DT_MENUS function is called recursively.

"Call Value" - The "Call Value" may be one of multiple items to be used by the "call type". The following table illustrates the required contents of the "call value" based upon the value of the "call type" entry:

Call Type	Call Value
EXECL	Valid executable program
SYSTEM	Valid shell command or executable object
CALL	Valid function name(defined in DTSFUNCT table)
RETURN	Return integer return value
CALLMENU	Valid MENU reference name

RELATED FUNCTIONS:

- DTPMENU - parsing function
- DT_MENUS - Primary menu function.
- DT_MNUCK - Called by the DT_MENUS function.

TYPEDEF:
DTTMENUS

```
dt_typdf.h
/******************************************************************************/
/* MENUS definitions */
#ifdef DTKMENUS
typedef struct {
COUNT num;          /* menu number */
COUNT imageno;      /* image number */
COUNT inputfld;     /* input field no */
COUNT cursfld;      /* last field that cursor was on */
COUNT comptyp;      /* compare type */
COUNT calltyp;      /* type of menu call */
TEXT *calltxt;      /* call text */
TEXT *comptxt;      /* compare input field text */
} DTTMENUS;

#endif
/******************************************************************************/
```

EXAMPLES:

```
                              Conventional Menu

IMAGE(1)  {LSTFLD_ADVANCE}
@DATE                         FairCom                              @TIME
                           d-tree Catalog


                                @CWD

                          1. Catalog
                          2. Table Dictionary
                          3. Column Dictionary
                          4. Index Dictionary
                          5. Segment Dictionary
                          6. Program Dictionary
                          7. Another Menu

                     Option: __

FIELD(1)
/* Symbol Name  Input Attribute  Output Attribute  Input Order          */
    option          NONE             NONE               1


MENU(1)
USES_IMAGE(1)
/*    Call Criteria       Type of Call        Call Value */
       option=1            SYSTEM              dtcatlog
       option=2            EXECL               dt_cattd
       option=3            RETURN              3
       option=4            CALL                menu2
       option=5            EXECL               dt_catsd
       option=6            RETURN              6
       option=7            CALLMENU            menu1
```

```
                         LOTUS STYLE MENU
IMAGE(2)  {INPUT_ADVANCE=CR}
    +

     _______      _______      _______      _______
                                                          +
FIELD(2)
/* Symbol Name  Input Attribute  Output Attribute  Input Order          */
    option1       NOCHANGE          INPUTRI           1
    option2       NOCHANGE          INPUTRI           2
    option3       NOCHANGE          INPUTRI           3
    option4       NOCHANGE          INPUTRI           4
DEFAULTS(2)
/* Symbol Name    Type of defaults        Defaults value */
    option1          INIT                   Checks
    option2          INIT                   Receipts
    option3          INIT                   Adjust
    option4          INIT                   Quit
MENU(2)
USES_IMAGE(2)
/*     Call Criteria       Type of Call          Call Value */
     CURSOR=option1         RETURN                  1
     CURSOR=option2         RETURN                  2
     CURSOR=option3         RETURN                  3
     CURSOR=option4         RETURN                  4
```

```
                       POP_UP/PULL DOWN MENU

IMAGE(3)  {INPUT_ADVANCE=CR}  {POP_UP}  {BASE_ROW=4}
    +
     _______
     _______
     _______
     _______
            +

FIELD(3)
/* Symbol Name  Input Attribute  Output Attribute  Input Order          */
    option1       NOCHANGE          INPUTRI           1
    option2       NOCHANGE          INPUTRI           2
    option3       NOCHANGE          INPUTRI           3
    option4       NOCHANGE          INPUTRI           4

DEFAULTS(3)
/* Symbol Name    Type of defaults        Defaults value */
    option1          INIT                   Modify
    option2          INIT                   Add
    option3          INIT                   Delete
    option4          INIT                   Quit

MENU(3)
USES_IMAGE(3)
/*     Call Criteria       Type of Call          Call Value */
     CURSOR=option1         RETURN                  1
     CURSOR=option2         RETURN                  2
     CURSOR=option3         RETURN                  3
     CURSOR=option4         RETURN                  4
```

For a full discussion and illustration of applying menus see the TUTORIAL Session 6.

INTERNAL d-tree REFERENCE - DTKMENUS

7.13 PROMPT - Data Base Access Prompt

DESCRIPTION:

The "PROMPT" ability provides a manner to define a method of access into a data base. Defining the desired Access to the data base necessitates the following: input from the user, a key or data file number determination; construction of a "target" value (TFRMKEY in c-tree) and a "significant length" to use in the access. The DT_PRMPT function provides a means by which the programmer can accept data from the user which will be used to initialize four useful work variables: key number, scan number, "target" value and the target's "significant length". These variables are then ready to use in the access call of choice (ie: EQLREC, FRSSET, GTEREC..etc).

SYNTAX:

PROMPT(reference_name)
    USES_IMAGE(image_reference_name)
/* key symbol name   scann name   fields for target   prefix */

```
PROMPT(master)
 USES_IMAGE(prompt)
/* key symbol name      scan name     fields for target   prefix */
     smcustidx           master          custnum             C
     smcustidx2          master2       custnam contact
     NONE                master3          option
```

"**reference_name**" - The "reference_name" is required and must uniquely identify this PROMPT definition section. No prerequisites or dependencies are placed upon this "reference_name".

Note: Before proceeding to define the associated parameter entries, it is first necessary to have a grasp upon the concepts of how the PROMPT ability works. Within your program the function DT_PROMPT is called. Its purpose is to do the following tasks:

- 1. Display an IMAGE.
- 2. Accept input from the IMAGE.
- 3. Analyze the user input with PROMPT definition.
- 3. Initialize the following associated variables:
  key number - based on user input, proper key number is set.
  scan number - associated scan number is set based in input.
  target value - field concatination and TFRMKEY builds target.
  target significant length - sets sig len for use in c-tree's SET calls.

These variables may then be used within a file access routine to retrieve the appropriate data record.

The DT_PRMPT function will first display the referenced IMAGE and accept input from the user. It will then evaluate which criteria line matches the data entered by the user, and initialize the key and scan number work variables using the associated values. Note that multiple fields may be entered in a single "fields for target". The same field may be used in more than one criteria line as long as each line defines a unique situation. Example: consider three fields on the input screen: FIELDA, FIELDB, and FIELDC. You may define the access as follows:

```
/* Key symbol    Scann     Fields for Target */
       KEY1      SCAN1     FIELDA
       KEY2      SCAN2     FIELDA FIELDB
       KEY3      SCAN3     FIELDB FIELDC
       KEY4      SCAN4     FIELDA FIELDB FIELDC
```

If only FIELDA is entered, use KEY1. If FIELDA and FIELDB is entered, use KEY2. FIELDB and FIELDC only, must be entered to use KEY3, and if all three fields have data the fourth key will be used. This feature provides greater flexibility in the selection process. The field(s) entered will be used to build the target. If a prefix exists, it is added to that target. Finally, the target's significant length is determined and placed into the corresponding work variable. All four work variables are then returned to the calling function.

> **NOTE:** For further explanation of how the PROMPT ability and the related DT_PROMPT function work, refer to the example at the end of this ability description.

"USES_IMAGE" - The "USES_IMAGE" keyword must be present and is used to identify the IMAGE that will be displayed and accept input . This identification is accomplished via the **"image_reference_name"**. The "image_reference_name" must be present and must reference a previously defined "IMAGE" ability section. The field symbols used in the "fields for target" definition must be found within the "IMAGE" referenced.

The following group of entries establish the values used to initialize the work variables based upon the data entered by the user.

"key symbol name" - The "key symbol name" is required and identifies the key symbol used to initialize the key number work variable. This key symbol must be a valid index symbol name found in either a c-tree parameter file or incremental structure. A special entry of "NONE" is allowed which results in a key number value of 0. This can provide a special indication to the programmer to provoke sequential or relative record number access.

"scann name" - The "scann name" is required and contains the reference name identifying the SCAN number to be used to initialize the scan number work variable. This allows the programmer the option to provoke a "scan" into the data base if the desired access fails. The scan is controlled by the programmer (a call to DT_SCANN). DT_PROMPT function only provides the associated scan number to use. See SCAN ability.

"fields for target" - The "fields for target" is required and must contain valid DODA symbolic name(s) which identify the data field(s) used to determine the proper prompt criteria. These fields are then used to form the "target" value. These entries must match the symbolic names contained within the "FIELDS" ability definition associated with the "IMAGE" ability referenced by the current "USES_IMAGE" "image_reference_name" entry.

"prefix" - The "prefix" entry must be a literal one and will be used to prefix the constructed target. This entry is optional and typically will be blank.

RELATED FUNCTIONS:

- DTPPRMPT - Parsing function.
- DT_PRMPT - Prompt routine. Provides key number, scan number, target, significant length of target for accessing a c-tree file.

TYPEDEF:
DTTPRMPT

dt_typdf.h

```
/*************************************************************************/
/* PROMPT definitions */
#ifdef DTKPRMPT
typedef struct {
COUNT   num:            /* prompt number */
COUNT   imageno:        /* image number  */
COUNT   scanno:         /* associated scann number */
TEXT *string:           /* prefix        */
} DTTPRMPT:

#endif
/*************************************************************************/
```

EXAMPLE:

The following script illustrates how the PROMPT ability may appear within a d-tree script.

```
IMAGE(prompt)   {INPUT_ADVANCE=1}
@DATE                      FairCom                        @TIME
                       Customer Master

Enter Customer Number:  _____
or
Enter Customer Name:    __________________________
or
Enter Record Number:    ______


                       Press ESC ESC to EXIT


FIELD(prompt)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  */
     custnum       NONE              NONE               1
     custnam       NONE              NONE               2
     number        NUMERIC           NONE               3
```

```
PROMPT(master)
 USES_IMAGE(prompt)
/* key symbol name   scann name    fields for target    prefix */
     smcustidx          master        custnum
     smcustidx2         master2       custnam
     NONE               master3       number
```

Below is a sample of how the DT_PROMPT function may appear within your program.

```
my_function()
{
COUNT prompt;
COUNT keyno;
COUNT scanno;
TEXT target[128];
COUNT targsiglen;
prompt = DT_INAME("prompt");
DT_PRMPT(prompt,&keyno,&scanno,target,&targsiglen);.
}
```

A good example of how the DT_PRMPT function is used in conjunction with the DT_SCANN and DT_EQREC functions is found within the delivered program source file DT_SCORE.C.

INTERNAL d-tree REFERENCE - DTKPRMPT

7.14 RTREE - Report Front-End

DESCRIPTION:

The "RTREE" ability provides a simplified method of building front end user interfaces to r-tree report programs.

SYNTAX:

RTREE(reference_name) ... see below

```
RTREE(my_report)
USES_IMAGE(my_report)          /* report prompt image */
USES_SCRIPT(ex_rtree.rts)      /* base r-tree script for report */
REPORT_PROGRAM(ex_rtree.exe)   /* r-tree report program */
RUN_IMAGE(my_message)          /* screen to display once report starts running*/
CALL_TYPE(MEMORY)              /* report in memory call */
CALL_TYPE(EXECL)               /* execl call */
CALL_TYPE(SYSTEM)              /* system call */
CALL_TYPE(SUBMIT)              /* submit to backgroup process */

/* r-tree  criteria Substitute */
/* keyword fields   String     */
   SEARCH  NONE     FILE "ARAY.DTA" ALL
           option1  FILE "ARAY.DTA" USING KEY KEY1 [ "{option1}"
           option2  FILE "ARAY.DTA" USING KEY KEY1   "{option2}" ]
           option1
           option2  FILE "ARAY.DTA" USING KEY KEY1 [ "{option1}" "{option2}" ]
   SELECT  NONE     ALL
           option5  (balance>0.00)
   VIRTUAL NONE     dev INTZ Z 1
           option6  dev INTZ Z "{option6}"
   SORT    NONE     LEAVE_OUT
           option7  NO_MOD "{option7}"
```

The "**reference_name**" must be a unique name identifying this particular RTREE ability section.

In session 6 of the tutorial we discussed step-by-step how to use d-tree to help in building an r-tree report. We also discussed how to use the RTREE ability and how it relates to the other pieces of r-tree report generation. You may find it helpful to review that particular session of the tutorial for further assistance in applying this ability.

The basic purpose of the RTREE ability is to assist in building the related specifications for front-end prompts to an r-tree report program. A typical front-end prompt for "SEARCH" range criteria (index key range to print , i.e. To and FROM Customer Number), "SELECT" criteria (i.e. Balance > 0.00), or how a report is to be "SORT"ed. The RTREE ability determines how the report is to be executed and allows the definition of an IMAGE that can be used as a "status" presentation to the user as the report is running.

Before proceeding step-by-step through the logic of how d-tree uses this ability, there are a few prerequisites. First, a base r-tree script must already exist. (See the r-tree reference manual for specific r-tree information and session 6 of the tutorial for information on using d-tree to assist in building this script.) Next, a C program, either user-written or generated by the CATALOG, which will use the r-tree script to perform the actual report generation, and finally a program that provides the user interface. This approach splits the process into two programs. A "prompt program" which contains the DT_RTREE function call which displays the prompt to the users, performs the proper substitutions to a base r-tree script and passes that script to a "report program". The "report program" contains the r-tree function call "report". The CATALOG program provides an easy way to create both these program's source specs. This "prompt program" and the DT_RTREE function is where we will focus our attention next.

The final goal of the DT_RTREE function is to generate an r-tree script, to be used by the called report program, which will produce the desired report based upon input accepted from the prompt screen.

To reach this goal DT_RTREE performs the following tasks:

- Display prompt IMAGE (USES_IMAGE).
- Accept user input.
- Merge base r-tree script (USES_SCRIPT) with RTREE script substitution definition based upon user input from the prompt screen.
- Display report "status" presentation to the user while report is running (REPORT_IMAGE).
- Call report program (USES_PROGRAM) (CALL_TYPE).

DT_RTREE, the primary RTREE ability function, will first display the IMAGE referenced by the USES_IMAGE entry and accept input from the screen. It then begins reading the pre-existing base r-tree script, as referenced by the USES_SCRIPT entry. As it reads the base script it searches for any r-tree keywords (SEARCH, SELECT, VIRTUAL and SORT) that were defined within the RTREE ability definition in the d-tree script. If one of these keywords are encountered it will then determine, based upon the user input values and the criteria from the d-tree script, the proper substitution. Simply stated, we are reading in one text file, and writing to a new (work) text file. As we do these writes we determine if there are any definition lines from the d-tree script that we want to insert (substitute) into the resulting script. The resulting script has a default name of DTRTS.BAK. We can better explain this merging process by using an example to illustrate.

The first illustration is the prompt screen (USES_IMAGE) to be displayed.

```
@DATE                         Customer Report Prompt                    @TIME
                                     FairCom

RANGE:   Customer Number:
         From: ______                      To: ______
            or
         Customer Name:
         From: _________________________   To: _________________________
         (To report on ALL customers, leave this section blank. If you
         wish start at the beginning leave the "From" field blank. If
         you want to report to the end, leave the "To" field blank.)


SELECT:  Print Accounts: __________ (Default: All)
         (=,<,>  +  amount) (i.e.: <> 0.00)

OUTPUT:  Send Output To Device: __  (Default: 1, Printer #1)
         (1 = Printer #1; 2 = Printer #2; 3 = Screen; 4 = Disk File)

SORT:    Sort (Y): _ (Default: No Sort)
```

This illustration is the corresponding RTREE ability section from the d-tree script.

```
RTREE(cust_report)
USES_IMAGE(cust_prompt)       /* report prompt image */
USES_SCRIPT(cust_rpt.rts)     /* base r-tree script for report */
REPORT_PROGRAM(cust_rpt.exe) /* r-tree report program */
RUN_IMAGE(run_msg)            /* screen to display once report starts running*/
CALL_TYPE(SYSTEM)             /* system call to report progam*/


/* r-tree  criteria Substitute */
/* keyword fields   String     */

   SEARCH  NONE     FILE "CUST.DTA  ALL
           option1  FILE "CUST.DTA" USING KEY KEY1 [ "{option1}"
           option2  FILE "CUST.DTA" USING KEY KEY1    "{option2}" ]
           option1
           option2  FILE "CUST.DTA" USING KEY KEY1 [ "{option1}" "{option2}" ]

           option3  FILE "CUST.DTA" USING KEY KEY2 [ "{option3}"
           option4  FILE "CUST.DTA" USING KEY KEY2    "{option4}" ]

           option3
           option4  FILE "CUST.DTA" USING KEY KEY2 [ "{option3}" "{option4}" ]

   SELECT  NONE     ALL
           option5  (balance{option5})

   VIRTUAL NONE     dev INT2 2 1
           option6  dev INT2 2 {option6}

   SORT    NONE     LEAVE_OUT
           option7  NO_MOD "{option7}"
```

In this example we have seven input fields used in our script, option1...option7. The four sections found on the input prompt screen, RANGE, SELECT, OUTPUT and SORT, relate to the four RTREE keywords SEARCH, SELECT, VIRTUAL and SORT, respectively. DT_RTREE will use the data entered by the user on the prompt screen to determine the proper entry for each keyword in the resulting r-tree script. In our example, DT_RTREE must first determine the proper SEARCH entry. If the user does not enter any RANGE information, the first criteria field entry in the SEARCH (d-tree script) section is applicable and its associated "substitute string" is entered into the r-tree script. If the user enters data only into the first input field, option1, the second line of the SEARCH (d-tree script) section would apply and its "substitute string" would be written to the r-tree script. Note the syntax used following the "KEY1" entry on that line in the d-tree script. If you are familiar with r-tree, the characters used in the syntax of the RTREE ability should be very familiar. The left square brace ("[") denotes that the following information is the beginning of the lower limit of the range. The right square brace ("]") denotes that the following information is the end of the upper limit of the range. If the user enters data in both RANGE input fields (option1 and option2) the fourth entry of the SEARCH (d-tree) section would apply and its corresponding "substitute string" would be written to the r-tree script. Thus, a lower and upper range limit would be established and the corresponding values would be substituted. The same logic applies to the Customer Name range fields, option3 and option4. DT_RTREE will continue down through the script until the first true condition is met for each keyword. Therefore, if the user were to enter data in all four RANGE fields only the fourth entry would be used.

Data entered by the user can be inserted into the "substitution string" by placing the "field symbol" within {} in the string.
(ie: SEARCH FILE A USING_KEY KEY1 [{field_symbol_1} {field_symbol_2}]

Since DT_RTREE will take whatever the user enters and substitute it into the proper r-tree script entry, we can allow the user to enter their own SELECT logic on the prompt screen. If they elect not to enter anything, it will insert the "NONE" "substitute string" into the r-tree script which, in this example, will print ALL records within the SEARCH range.

The next section VIRTUAL provides for the definition of virtual fields. In our example we are using a virtual field to represent the output device which will produce our report. If the user leaves this field blank DT_RTREE will find a match on the "NONE" field entry and write the corresponding "substitute string" , to the r-tree script. If a value is entered, this example will insert the value into the matching "substitute string" which is written to the r-tree script.

The same type of logic is performed for the SORT keyword section. Note, however, one important d-tree keyword (LEAVE_OUT) illustrated by the SORT section. This keyword will direct DT_RTREE to simply "leave out" the r-tree keyword in the resulting script. If the NONE criteria is true, no SORT entry will be made.

Once all substitutions have been made and the script has been built, DT_DTREE will display the specified RUN_IMAGE. The following is an example RUN_IMAGE.

```
@DATE                            Customer Report                           @TIME
                                    FairCom



                        Your report is now being processed.


                               One Moment Please ...
```

KEYWORD ENTRIES:

USES_IMAGE(prompt_image_reference) - The "**USES_IMAGE**" entry identifies the prompt IMAGE definition to be displayed. The "**prompt_image_reference**" must match the reference name of the prompt IMAGE to be displayed.

USES_SCRIPT(base_script) - The "**USES_SCRIPT**" identifies the previously defined r-tree script used by the function DT_RTREE. The "**base_script**" is the filename containing the r-tree script. Traditionally these files use RTS extensions.

REPORT_PROGRAM(program_name) - The "**REPORT_PROGRAM**" identifies the program which will be called to generate the final report based upon the constructed r-tree script. The "**program_name**" must be the name of an executable program which accepts a script name as a parameter and calls the r-tree "report" function. (note: the catalog will create this for you).

CALL_TYPE(call_keyword) - The "**CALL_TYPE**" identifies the "type of call" to be used by the "prompt program" to pass control to the report process. Only one call type is permitted per RTREE section. The "**call_keyword**" must be one of the following valid CALL_TYPEs. (see dt_rtree.c code)

- **EXECL** Execute this report as a separate process with an "execlp" call.
- **SYSTEM** Invoke a system call. After execution is complete, control will be returned to the calling program.
- **SUBMIT** Submit for background processing. Multi-tasking environments only.
- **MEMORY** The r-tree report function is being called directly from the prompt program.

  NOTE: The MEMORY CALL_TYPE assumes that only one program is being used. Report execution is in the same program as the prompt program. (see #define in dt_ctree.c for the inclusion of the "report" function).

/* r-tree	**criteria**	**Substitute**	***/**
/* keyword	**fields**	**String**	***/**

"**r-tree keyword**" - The "r-tree keyword" is one of the following:

- **VIRTUAL -** Identifies any necessary virtual field definitions.
- **SEARCH -** Identifies the range of records to process.
- **SELECT -** Identifies any selection criteria to be applied.
- **SORT -** Identifies any sort specifications.

See the r-tree reference manual for further information on these keywords

"**criteria fields**" - The "**criteria fields**" relate to what input fields on the prompt screen have been entered by the user. Entries must be either the keyword "NONE" or valid field symbolic names. (Combinations of field symbolic names may be used by placing them directly beneath one another).

"**substitute String**" - The "**substitute string**" is the string of data to be written to the r-tree script when a true "criteria fields" condition is met.

RELATED FUNCTIONS:

- DTPRTREE - Parsing function.
- DT_RTRE2 - r-tree interface secondary substitution function.
- DT_RTREE - Primary r-tree front end function.

TYPEDEF:

DTTRTREE

```
dt_typdf.h
/********************************************************************************/
/* RTREE interface definitions */
#ifdef DTKRTREE
typedef struct {
COUNT    num;              /* rtree definition number */
COUNT    imageno;          /* image number  */
COUNT    type;             /* interface type */
COUNT    rtkeywd;          /* rtree keyword reference number */
TEXT    *script;           /* base r-tree script name */
TEXT    *string;           /* substitute string */
TEXT    *program;          /* program to run */
} DTTRTREE;

#endif
/********************************************************************************/
```

INTERNAL d-tree REFERENCE - DTKRTREE

THIS PAGE LEFT BLANK INTENTIONALLY

7.15 SCAN - Scan or browse data base

DESCRIPTION:

The "SCAN" ability provides for browsing (scanning) through data files.

SYNTAX:

SCAN(reference_name) {IMAGE_OUT=?} {IMAGE_ROLL=?} {IMAGE_INP=?}

```
SCAN(master)  {IMAGE_OUT=fixed_img {IMAGE_ROL=roll_img} {IMAGE_INP=input_img}
              {USE_SETS=Z}
```

"reference_name" - The reference_name is the unique identifier for each SCAN. No prerequisites or dependencies apply to this ability reference name.

This ability is directly related to the d-tree function DT_SCANN. This single function performs many very useful tasks. The calling program must pass this function the following values:

- scan number
- key number
- target
- significant length of the target

  Note that all of these values may be retrieved by the d-tree function DT_PRMPT.

DT_SCANN will then perform the following tasks:

- project a fixed heading screen (IMAGE_OUT)
- access the data file starting at the point indicated by the target
- read records
- project records in a 'rolling' area on the screen (IMAGE_ROL)
- control cursor handling capabilities of scrolling up and down through the data records
- handle the editing of data records while displayed in the 'scrolling' portion of the screen
- handle data record selection
- accept input from either the first or second image(IMAGE_INP)

All of these tasks are neatly packaged in this single powerful function.
The DT_SCANN function will return a value greater than zero if no record has been selected during the scan process. Otherwise, it will return a value of zero meaning a record has been accessed.

KEYWORDS:

IMAGE_OUT=image_reference_name

The "**IMAGE_OUT**" keyword identifies the IMAGE to be displayed first. The "**image_reference_name**" must match that of a previously defined IMAGE ability section. This IMAGE is normally a fixed header type of display. It normally contains titles and other constant header information but can also contain I/O fields, such as a selection input field.

IMAGE_ROL=image_reference_name

The "**IMAGE_ROL**" keyword identifies the IMAGE which defines the rolling portion of the display where the records are to be presented. The "**image_reference_name**" must reference the name of a previously defined IMAGE ability section.

IMAGE_INP=image_reference_name

The "**IMAGE_INP**" keyword identifies the IMAGE containing any input field(s). The "**image_reference_name**" must match the image reference name of either the IMAGE_ROL or IMAGE_INP keyword entries. Thus, the image referenced must be a previously defined IMAGE ability section. Only one IMAGE_INP image reference name is permitted per SCAN ability section.

Since you may receive input from either the first or second IMAGE, there are two basic approaches to using the SCAN feature:

Method #1 is to establish the IMAGE_INP as the same IMAGE used by the IMAGE_OUT keyword entry. This approach will display a static or fixed header screen which should contain a single input field (normally at the bottom of the display). Next, display the records using the scrolling or IMAGE_ROL portion of the screen. Each record should be preceeded by a unique number, the d-tree global variable "counter" must be defined on IMAGE_ROLL screen . The user may select a record by entering the number corresponding to the desired record in the input field on the fixed IMAGE. Reference the TUTORIAL for further illustration of this method.

Method #2 is to indicate that the IMAGE_INP is the same IMAGE as the IMAGE_ROL. This is done by using the same image reference names. This will then allow the cursor to move freely through the data fields within the scrolling portion of the screen. You may edit the data and the disk file will be updated as well. This provided the ability to maintain multiple data records on the screen at one time. Multi-user aspects such as record locking and multi-user interference are not strongly considered at this time in the logic. Please review the dt_scann.c file to see the read and write logic if multi-user considerations are necessary.

USE_SETS{=number}
The USE_SETS option directs d-tree to use c-tree SET logic (FRSSET, NXTSET). In brief, FRSSET logic allows access to a group of qualifying records within a data file based upon the target issued and a significant length. For example, if the target issued was 'SMITH' with length 5, only the records beginning with 'SMITH' in the specified field would qualify for selection. The optional suffix "=number" will direct d-tree to use only that number of bytes for significant length of the target. Otherwise, the length of the target is used for the significant length.
For example: Given: target = 'C101' and {USE_SETS=2}
The data file would be accessed at the first occurence of the target 'C101', or closest point thereto if there was no find on the target. Only records therafter containing the value 'C1' in the first two positions of the target would be retrieved for display to the IMAGE_ROL portion of the screen.

RELATED FUNCTIONS:

- DTPSCANN - Parsing function
- DT_SCANN - Primary SCAN function for c-tree data file.

TYPEDEF:
DTTSCANN

```
dt_typdf.h
/***********************************************************************/
/* SCAN definitions */
#ifdef DTKSCANN
typedef struct {
COUNT    num;              /* scan number */
COUNT    imgout;           /* header image number  */
COUNT    imgrol;           /* rolling image number  */
COUNT    imginp;           /* input image number  */
COUNT    useset;           /* use c-tree FRSSET logic
                           /* values 0 = do not use sets */
                           /*       -1 = use provided target sig len */
                           /*      > 0 = the sig length to use for sets */
COUNT    rollines;         /* number of lines to roll */
} DTTSCANN;

#endif
/***********************************************************************/
```

EXAMPLE:

```
IMAGE(heading)  {LSTFLD_ADVANCE}  {FRSFLD_BACKUP}
@DATE                          FairCom                            @TIME

                            Customer Master
Select    Customer Name                     Number          Amount






                       Enter Desired Option: __
                        Press ESC ESC to EXIT

FIELD(heading)
/* Symbol Name    Input Attribute  Output Attribute  Input Order  */
     Option          NONE              NONE                1

IMAGE(rollpart) {NO_CLS} {LSTFLD_ADVANCE} {FRSFLD_BACKUP} {BASE_ROW=4}
          __________________________     _____      ______.__
```

```
FIELD(rollpart)
/* Symbol Name    Input Attribute  Output Attribute  Input Order  */
     counter        NONE             NONE              1
     custnam        NONE             NONE              2
     custnum        NONE             NONE              3
     custbal        NONE             NONE              4

SCAN(master)  {IMAGE_OUT=heading} {IMAGE_ROL=rollpart} {IMAGE_INP=heading}
```

This IMAGE is displayed first.

This IMAGE shows data base rcds in a scollable screen region.

This IMAGE accepts user input.

INTERNAL d-tree REFERENCE - DTKSCANN

7.16 SUBFILE - Related groups of records

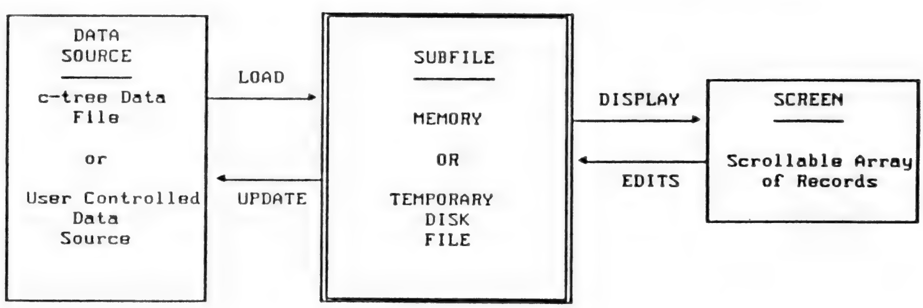
DESCRIPTION:
The "SUBFILE" ability provides a means to process a related group of records in a temporary work space (memory or temporary disk file).

SYNTAX:
SUBFILE(reference_name)

```
SUBFILE(master)
SFL_IMAGE(trans)
SFL_RECORDS(32)
SFL_LINES(16)
SFL_TITLE(transtitle)

SFL_TARGET
/* key symbol name         fields for target         prefix */
     dt_cdseq_idx          td_fil td_version

SFL_MAP
/* parent field            child field          length */
     td_fil                   cd_fil
     td_version               cd_version
     SFL_SEQ                  cd_fldseq

MUSTHAVE
     cd_fldnam
```

"reference_name" - The "reference_name" entry uniquely identifies this SUBFILE ability definition section. No dependencies or requirements are placed upon this name.

The additional d-tree script entries used to define a subfile are best presented while discussing the subfile concept. Consider the following illustration:



Physically, a SUBFILE is either an allocated block of memory or a temporary disk file. If an allocated block of memory is used, all subfile records reside in memory, and may thus contain only a limited number of records. If you elect to use a temporary disk file, you may have a subfile limited by only the amount of

available disk space. Processing speed is the advantage of the memory resident subfile. Unlimited record capability makes the temporary disk file approach more applicable in some cases.

There are basically four areas which must be addressed when using a subfile:

- **Initializing** the subfile.
- **Loading** the subfile.
- **Processing** the subfile.
- **Updating** from the subfile.

We must initialize an area for our subfile, either memory or disk. This allocated area must then be loaded, either from an existing disk file or another source controlled by user logic. The data residing in the subfile may then be processed, typically using screen I/O or another user program defined method. After processing is complete, the updated data can then be written back to the original disk file or other user program supplied source. Let's take a closer look at each step.

Initialize a Subfile

First, initializing a subfile. To initialize a subfile d-tree must know two things:

- Type of Subfile Work Area (memory or disk).
- Record Size (related c-tree file or user defined group of fields defined in the DODA).

Type of Subfile Work Area

You may select one of two places to build a subfile:

- **Allocated Memory Block** - SFL_RECORDS(number_of_record)
  The determination if a subfile is in memory or a temporary disk file is made by the SFL_RECORDS entry. If a fixed number of records is given (ie: SFL_RECORDS(48)) the memory subfile is assumed, defined to hold that many records (ie: 48 records). d-tree will calculate the amount of memory to allocate by multiplying the number of records by the maximum record size. The record size is described below.

- **Temporary Disk File** - SFL_RECORDS(1)
  If the SFL_RECORDS entry is a one (1), the subfile is defined as a temporary disk file (note: there is no such thing as a subfile that only holds one (1) record. The concept of a subfile is to process multiple records).

Record Size -The second requirement to initializing a subfile is how the size of the records within the subfile will be determined. This may be done by either defining the subfile to have a direct association with a c-tree file or by defining an association with a group of DODA fields.

- **Association With c-tree File** - SFL_TARGET
  d-tree determines the record length via the SFL_TARGET keyword entry. Using the key symbol name parameter entered with this keyword, d-tree has access to the c-tree file definition. The record length found in this c-tree file definition will be the record length used for the subfile. This forms a direct association between the subfile and the c-tree disk file. The SFL_TARGET keyword will be explained in more detail in the subfile processing section.

- **Group of Fields in DODA** - SFL_BOUNDARY
  The second option is establishing an association with a group of fields defined in a DODA. The SFL_BOUNDARY entry defines the first and last fields of a group of fields defined in a DODA. This group of fields may be any fields you may want to group together to form a subfile. The entries in the DODA must be consecutive, allowing a top and bottom boundary to be defined. Using the identified field definitions from the DODA, d-tree is able to determine a subfile record length.

Load a Subfile
Once space for our subfile has been allocated and initialized, we must fill it with data. If there is no direct association with a c-tree file, the user controls the load logic (DT_SFLNW or DT_SFLRW). If there is a direct association with a c-tree file, d-tree provides functions to perform the load process. d-tree needs the following information in order to load the subfile:

- Source of data.
- Key for file access (if disk file used).
- Target for access (if disk file used).
- Destination of data.

All of the above requirements are provided via the "**SFL_TARGET**" keyword. By providing the "**key_symbolic_name**" it not only informs d-tree of the file to access (given a key name, d-tree will know its corresponding data file) but also that file's key for access. The target for access is found in the "**fields_for_target**" entry.

The contents of the fields or partial fields, represented by the symbolic names entered, will be concatenated together to form the "target" used to access the file via the previously mentioned key. If a "**prefix**" is defined, it will be given an address when the "target" is formed. Given the index and the "target" a subfile can be loaded with records with c-tree's FRSSET/NXTSET functions.

Subfile records are ALWAYS processed as fixed-length records. However, the source file definitions may be fixed or variable length. As the subfile is being loaded from its defined source, all variable length records are unpacked into the subfile. **NOTE:** The loading of subfiles, as defined in this section, pertain solely to subfiles initialized with a direct relationship to a c-tree disk file, using the SFL_TARGET keyword. Using the optional method of initializing a subfile with an association to a group of fields defined within a DODA, using the SFL_BOUNDARY keyword, requires the user program to be responsible for the load procedures.

Processing a Subfile

After loading the subfile, we are free to process the subfile's data. d-tree provides functions for processing this related group of records. A few of these are:

Standard Read/Write Processing

- first - DT_FSSFL
- last - DT_LSSFL
- next - DT_NXSFL
- previous - DT_PVSFL
- read - DT_EQSFL
- write - DT_SFLRW
- add - DT_SFLNW

Standard I/O processing may use the d-tree functions but must be maintained by the user program.

Screen I/O Processing

Screen processing is a user's window to the subfile. As the user edits data on the screen, they are editing the actual subfile. The screen I/O processing may be controlled from within the d-tree script. For our discussion, we will divide the screen processing into two obvious classifications, input and output. First, output.

To display the records, an associated IMAGE must be defined. This IMAGE is a scrollable region on the screen and is 'tied' to the subfile by using the **'SFL_IMAGE'** keyword. Static headers may be defined to identify column headings pertaining to the data displayed in the subfile. This header or title IMAGE is identified by the **'SFL_TITLE'** keyword.

One other parameter which must be defined for output is how large of an area on the screen should be used to display the subfile, the whole screen or just a small window.The width of the subfile display is controlled by the SFL_IMAGE but the length is controlled by the 'SFL_LINES' keyword entry. Note that this entry does not include the number of lines required by the header portion of the display, SFL_TITLE, only the number of lines to be actually used by the data records. This entry must be large enough to contain at least one record and should be divisible by the number of lines required by a single record. This number should also divide evenly into the SFL_RECORDS entry (assuming the SFL_RECORDS is not 1). The maximum lines any subfile record can be is the screen size (typically 24 lines) .

Input screen processing is nicely controlled by d-tree. This includes cursor controls, page up, page down, maintaining input fields, etc. When viewing multiple screens of records, you have the ability to control when the subfile image will scroll to the next page of records. The **SFL_ATTR** keyword with the **NO_ROLL_CR** attribute option will inhibit the subfile from scrolling when a carriage return is issued while the cursor is positioned on the last field of the last record on the screen.

Updating From a Subfile

If the subfile was initialized and loaded from an associated c- tree disk file, d-tree provides functions to handle the update procedures. The method of update used by d-tree is the 'wash' method. All original records loaded into the subfile are deleted (DT_DLSFL) from the original c-tree file and the edited records in the subfile are loaded (added DT_ADREC) back into the c-tree disk file. Consider the following update flow:

- 1) First all records in the c-tree file that were loaded into the subfile are deleted from disk.
- 2) Read a subfile record.
- 3) Any data defined to be copied into (mapped) each subfile record is now copied. The definition of data to be mapped is provided by the **SFL_MAP** keyword. Simply place the "parent" or source field symbol name along with the "child" or destination field symbol within the SFL_MAP section in the d-tree script.
- 4) The subfile record is checked to see if is "worthy" to be written to disk. The definition used to determine this is provided by the **SFL_MUSTHAVE** keyword. A record is "worthy" to be written to disk is it passes the "must have" check. Within the SFL_MUSTHAVE section define the fields that "must have" data in order for this record to be "worthy". If the record passes this check it is written (added_ to the c-tree file.)
- 5) Repeat steps 2 thru 4 for each subfile record.

KEYWORDS:

- **SFL_IMAGE(image reference)** - The "SFL_IMAGE" keyword identifies the associated image by which the subfile records will be displayed.
- **SFL_TITLE(title reference)** - The "SFL_TITLE" keyword identifies the static header image to be displayed over the subfile records.
- **SFL_RECORDS(number)** - The "SFL_RECORDS" keyword sets the maximum number of records which will be buffered in the sub- file. An entry of '1' designates an unlimited number of sub-file records may exist within a disk file.
- **SFL_LINES(number)** - The "SFL_LINES" keyword identifies the number of lines to be used by the subfile records on the screen.
- **SFL_ATTR** - NO_ROLL_CR - The "SFL_ATTR" keyword and "NO_ROLL_CR" input attribute inhibit screen from paging to the next screen full of data when a carriage return is issued while the cursor is positioned on the last line of data. The cursor will 'wrap' to the top of the display.
- SFL_TARGET
  /* key symbol name   fields for target   prefix */
  The "SFL_TARGET" keyword and related entries are only to be used when loading a subfile from a c-tree disk file. This entry is used by d-tree to identify the c-tree file to be accessed, the key to the file, the record size to be used, the target for access, and any prefix to attatch to the target before attempting access.

  key symbol name - The "key symbol name" is the symbolic name of the index key of the c-tree disk file to be used to load the subfile.

  fields for target - The "fields for target" are symbolic field names, which when concatenated will construct the target to be used, via the key of the c-tree disk file, to load the the subfile. Multiple symbolic field names are allowed. To use only the first n bytes of a field, place the number of bytes to be used at the end of the end of its symbolic field name surrounded by parentheses. (i.e. cust_name(10)

  prefix - The "prefix" will be placed at the front of the contents of the constructed target before attempting to access the data. This must be a literal value.

- SFL_BOUNDARY
  /* doda first field   doda last field */
  The "SFL_BOUNDARY" defines the first and last fields of a group of fields within the DODA. This entry is only used when the subfile is to be loaded with data from a source other than a c- tree disk file. The fields to be used in the DODA must be contiguous. This is used by d-tree to determine the record size of the subfile.

The "doda first field" is the first field of a group of fields defined in the DODA.
The "doda last field" is the last field of group of fields defined in the DODA.

- SFL_MAP
  /* parent field child field length */
  The "SFL_MAP" option is used to copy data into subfile records just before they are updated to disk. For instance, if the master file has a field for customer ID and the subfile does too, this allows a change to the master field to automatically be updated to the subfile. Parent will be in master and child in subfile.
  EXAMPLE:
  The "parent field" is the field in the Master file which will contain the information to be copied.
  The "child field" is the field in the subfile which will receive the data.
  The "length" is the length of the data being mapped. If no length is specified the shorter of the two field lengths, "parent" or "child", is used.

- SFL_MUSTHAVE - The "SFL_MUSTHAVE" keyword identifies fields which are required to contain data before the record is written to disk. (Note: edits be placed upon these fields at input).

RELATED FUNCTIONS:

- DTPSUBFL - Parsing function.
- DT_SFCAD - Subfile Child Add Routine.
- DT_SFCLD - Load child subfile routine.
- DT_SUBFL - Maintain Subfile.
- DT_SFLOT - Subfile Out. Display the Subfile.
- DT_SFHLD - Subfile High Level Load.
- DT_SFLLD - Load Subfile-Low Level.
- DT_SFHDL - Subfile High Level delete.
- DT_SFLDL - Subfile low level delete-delete a group of c-tree records.
- DT_SFHAD - Subfile High level add.
- DT_SFLAD - Subfile Low Level Add Routine.
- DT_SFLRM - Remove temporary subfiles from disk.
- DT_FSSFL - Get first record in subfile.
- DT_EQSFL - Get record in subfile.
- DT_LSSFL - Get Last subfile record.
- DT_NXSFL - Get next record in subfile.
- DT_PVSFL - Get previous subfile record.
- DT_SFLRW - Subfile rewrite.
- DT_SFLNW - Add a new record to subfile.
- DT_EDSFL - Edit a subfile.

TYPEDEF:
DTTSUBFL

```
                         dt_typdf.h
/************************************************************************/
/* SUBFL definitions */
#ifdef DTKSUBFL

typedef struct {
TEXT    *sptr;             /* sfl memory block */
COUNT   datno;             /* subfile temp file datno (unlimited subfiles) */
TEXT    target[MAXLEN]:    /* target used to load sfl */
COUNT   tarsigln;          /* target sig length */
COUNT   noofrcds;          /* number of records in sfl */
COUNT   currcd;            /* current sfl record */
COUNT   currow;            /* current sfl row */
} DTTSUBSB;

typedef struct {
COUNT   num;               /* subfile number */
COUNT   imageno;           /* image number  */
COUNT   title;             /* title image number  */
TEXT    *prefix;           /* target prefix */
COUNT   maxrcds;           /* max number of records for subfile */
COUNT   sfllines;          /* total number of display lines for subfile */
COUNT   startdoda;         /* starting doda occurance number */
```

Example:

```
IMAGE(master) {LSTFLD_ADVANCE}
@DATE                         FairCom                         @TIME
File Name: _____________ File Description: _____________________
Validation Number: _____ System Name:      _____________
File Type: __________ Extension: ___ Mode: __ Rcd Len: _____ Indexes: ___

IMAGE(transtitle) {CLR_LINES=19} {BASE_ROW=5}
Field                     Field                          First Vlen
Name      Type Len  Dec   Description                      Field

IMAGE(trans) {NO_CLS} {LSTFLE_ADVANCE} {FRSFLD_BACKUP} {BASE_ROW=7}
_______ ___ ___  _  ____________________________    ___

FIELD(master)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  I/O Mask */
     td_fil         ALLCAPS               NONE               1
     td_desc        SCROLL ALLWORDCAPS    NONE               2
     td_version     NUMERIC               NONE               3
     td_system      FRSWORDCAPS           NONE               4
     td_type        ALLCAPS               NONE               5
     dxtdsiz        NUMERIC               NONE               6
     dfilmod        NUMERIC               NONE               7
     dreclen        PROTECT               NONE               8
    dnumidx         PROTECT                NONE               9

FIELD(trans)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  I/O Mask */
     cd_fldnam      NONE                  EOL                1
     cd_fldtyp      ALLCAPS  TABLE_IN     TABLE_OUT          2
     cd_fldlen      NUMERIC               NONE               3
     cd_dec         NUMERIC               NONE               4
     cd_desc        NAMECAPS              NONE               5
     cd_vlen        ALLCAPS               NONE               6

SUBFILE(master)
SFL_IMAGE(trans)
SFL_RECORDS(32)
SFL_LINES(16)
SFL_TITLE(transtitle)

SFL_TARGET
/* key symbol name        fields for target         prefix */
     dt_cdseq_idx         td_fil td_version

SFL_MAP
/* parent field           child field          length */
       td_fil                 cd_fil
       td_version             cd_version
       SFL_SEQ                cd_fldseq

  MUSTHAVE
       cd_fldnam
```

Internal d-tree Reference - DTKSUBFL

THIS PAGE LEFT BLANK INTENTIONALLY

7.17 TABLES - Alternate Data Representation

DESCRIPTION:

The "TABLES" ability establishes a cross-reference between data representation on disk and its appearance when displayed on the screen. This ability allows coded data from the data base to be presented in "alternate" form to the user. This "alternate" form can also be entered by the user and this ability will convert it to the "coded" form on disk.

SYNTAX:

TABLES(reference_name)
disk representation screen representation source field(s)

```
TABLES(trans)
/* disk representation   screen representation    source field */
         1                       C                  cd_fldtyp
         2                       CU                 cd_fldtyp
         3                       I                  cd_fldtyp
         4                       IU                 cd_fldtyp
```

"reference_name" - The "reference_name" must be a unique identifier for this TABLE definition section.

NOTE: Although multiple TABLE sections are allowed, d-tree will consolidate all TABLE definition sections into one master table, therefore, the same field cannot have more than one alternative representation.

The following describes what invokes the TABLE ability and how it works. The TABLE ability is directly associated with the TABLE_IN and TABLE_OUT input and output attributes defined in the FIELD section:

TABLE_IN - When a field has been defined with a TABLE_IN attribute, you are indicating that when a value is entered into this field the table should be checked to attempt to find a "screen representation" for this field that is equal to the value that was just entered. If one is found, the disk representation is then substituted into that field's address. This is very useful in presenting data in a more "user friendly" manner while storing a minimum number of bytes (coded data) into the data base.

TABLE_OUT -The TABLE_OUT is the opposite of the TABLE_IN and pertains to output. If a TABLE_OUT attribute is encountered on a specific field when it is being displayed, d-tree will search the TABLE for an entry for that specific field which has a "disk representation" value equal to the value of the field. If a match is found, the corresponding "screen representation" from the TABLE is displayed. This allows you to take "coded" data from disk and expand it into a more "user friendly" representation on the screen.

TABLE ENTRY - a single table entry contains the following:

- **"disk representation"** - The "disk representation" is the contents of the data field as it is to be stored on disk.
- **"screen representation"** - The "screen representation" is the contents of the data field as it is to be displayed or entered on the screen.
- **"source field(s)"** - The "source field" is the symbol name(s) representing the field(s) which contains an "alternative" definition.

RELATED FUNCTIONS:

- DTPTABLE - Parsing function
- DT_FLDIN - Input a field (Field In). When TABLE-IN attribute is considered
- DT_FLDOT - Display a field (Field Out). When TABLE-OUT is considered.

TYPEDEF:

DTTTABLE

dt_typdf.h

```
/************************************************************************/
/* TABLE interface definitions */
#ifdef DTKTABLE

typedef struct {
COUNT    num;            /* table number */
COUNT    fdodano;        /* doda field number */
TEXT     *disk;          /* disk representation of field */
TEXT     *scrn;          /* screen representation of field */
} DTTTABLE;

#endif
/************************************************************************/
```

Example:

The following example is taken from the DTCATLOG.DTS script. When using the DTCATLOG program and adding a new file, field types are entered as 'C' for char, 'CU' for char unsigned, etc. The program, via TABLES, will store a 1 to disk to represent the character 'C' and a value of 2 to represent the characters 'CU'.

```
IMAGE(trans)   {CLR_LINES=19} {BASE_ROW=5}
 Field                            Field                             First Vlen
 Name        Type  Len  Dec  Description                          Field

____________  ____  ___  _  ______________________________         ___

FIELD(trans)
/* Symbol Name  Input Attribute  Output Attribute  Input Order  */
    cd_fldnam       NONE              EOL                1
    cd_fldtyp   ALLCAPS TABLE_IN      TABLE-OUT          2
    cd_fldlen     NUMERIC             NONE               3
    cd_dec        NUMERIC             NONE               4
    cd_desc       NAMECAPS            NONE               5
    cd_vlen       ALLCAPS             NONE               6

TABLES(trans)
/* disk representation   screen representation    source field */
           1                       C                 cd_fldtyp
           2                       CU                cd_fldtyp
           3                       I                 cd_fldtyp
           4                       IU                cd_fldtyp
           5                       L                 cd_fldtyp
           6                       LU                cd_fldtyp
           7                       SF                cd_fldtyp
           8                       DF                cd_fldtyp
           9                       DA                cd_fldtyp
           10                      MO                cd_fldtyp
           11                      L                 cd_fldtyp
           12                      A                 cd_fldtyp
```

INTERNAL d-tree REFERENCE - DTKTABLES

THIS PAGE LEFT INTENTIONALLY BLANK

Function Reference

NAME
DT_ADDIT- Add one field to another given only DODA symbol names. (doubles only) (DT_WDODA.C)

TYPE
Data Management

DECLARATION
```
COUNT    DT_ADDIT(dsymb,ssymb)
TEXT     *dsymb;     /* desination symbol */
TEXT     *ssymb;     /* source symbol */
```

DESCRIPTION
Given two DODA symbolic names, add the value of the second to the first. This function uses DT_TDODA to get the addresses pertaining to the symbol names and then does the addition. The result is that the value at the destination symbols's address has now been incremented by the value at the source's address. This function assumes that the symbols being passed are of double type. Use this as an example if other types are needed.

RETURN
NORMAL RETURN
Returns a zero if successful.
ERROR RETURN
Returns a 1 if error.

EXAMPLE
```
if (DT_ADDIT("F1400a","F703a"))
        printf("Could not add the two values.");
```

SEE ALSO
DT_TDODA

REFERENCE NUMBER - 2B

NAME
DT_ADREC- Add a record to the c-tree data base. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_ADREC(datno)
COUNT     datno;          /* data file number to add record to. */
```

DESCRIPTION
Add a record to a c-tree file. This function uses the d-tree global record buffers pointed to by dt_sfp[datno] as the record buffer for the add. This function will do the following steps:

1) Pad the fields in the record buffer with the PADDING character from c-tree.
2) Execute uniformat logic if defined by c-tree.
3) Determine the type of file record is being added to: either fixed or variable length
4) If variable length file it will pack the record.
5) Enable record locking. LKISAM(ENABLE).
6) Execute c-tree's add record call. Either ADDREC or ADDVREC.
7) Free the address record. LKISAM(FREE).

RETURN
NORMAL RETURN
Returns a zero for successful add.
ERROR RETURN
Returns the c-tree isam_err value if add has failed.

EXAMPLE
```
if (err=DT_ADREC(datno))
        printf(target,"Unable to Add record c-tree error = %d",err);
```

REFERENCE NUMBER - 30

NAME
DT_ALDOD- Set the offset values for the field entries in a DODA and return either the record length for this group of fields or the number of fields in the record. (DT_ALIGN.C)

TYPE
DODA Management/Internal Structure Relationships

DECLARATION
```
UCOUNT   DT_ALDOD(dodaptr,nooffids)
DATOBJ   *dodaptr;     /* pointer to first field in record */
COUNT    nooffids;     /* number of fields in record */
```

DESCRIPTION
Given a starting DODA pointer that is considered to be the first field in a record structure, this function will figure out the offsets of each field and place the offset value in the fhrc field in the DATOBJ structure. If a number of fields is provided, the record length will be returned. If number of fields is zero, this function will return the number of fields it found in the DODA. This function utilizes the function DT_STALN to take into consideration the alignment restrictions of the hardware.

RETURN
If number of fields passed is zero returns number of fields found in the doda else if number of fields passed is non-zero.
Returns the record length of this group of fields.

EXAMPLE
```
rcdlen=DT_ALDOD(dodaptr,nooffields);
```

SEE ALSO
```
DT_STALN();        /* set alignment array */
DT_OFSET();        /* calc field offsets */
DT_RCDLN();        /* calc record length */
```

REFERENCE NUMBER - 1C

NAME
DT_CALCS- perform defined calculation. (DT_CALCS.C)

TYPE
Data Managment

DECLARATION
```
COUNT      DT_CALCS(calcptr,ivalue,fvalue)
DTTCALCS   *calcptr;     /* pointer to CALCS type definition */
LONG       *ivalue;      /* pointer to LONG result */
double     *fvalue;      /* pointer to floating point result */
```

DESCRIPTION
When a CALCS ability is parsed from a d-tree script, it is converted from infix to postfix form. A pointer to this postfix expression is stored in the CALCS structure. DT_EVALU is then called. Using a stack, DT_EVALU evaluates the postfix expression and performs the appropriate calculations giving the desired result.

RETURN
Always returns zero

EXAMPLE
```
DT_CALCS((DTTCALCS *)calcptr,&wlong,&wdouble);
```

REFERENCE NUMBER - 83

NAME
DT_CLEAR- Clear the Screen. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
COUNT DT_CLEAR()

DESCRIPTION
When DT_KEYBD is called , special control sequences are defined for both keyboard and screen. This function outputs to stdout the special sequence of characters that were defined in the TERMCAP file to clear the screen.

RETURN
Always returns a zero.

EXAMPLE
DT_CLEAR();

SEE ALSO
DT_KEYBD(); /* Initialize keyboard and Screen definitions from TERMCAP */

REFERENCE NUMBER - 83

NAME
DT_CLEOL- Clear Screen to the end of a line. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
COUNT DT_CLEOL()

DESCRIPTION
When DT_KEYBD is called, special control sequences are defined for both keyboard and screen. This function outputs to stdout the special sequence of characters that were defined in the TERMCAP file to clear the screen to the end of a line.

RETURN
Always return a zero.

EXAMPLE
DT_CLEOL();

SEE ALSO
DT_KEYBD(); /* Initialize keyboard and Screen definitions from TERMCAP */

REFERENCE NUMBER - 50

NAME
DT_CLRBK- Clear a Block on the screen. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
```
COUNT    DT_CLRBK(topcol,toprow,lstcol,lstrow)
COUNT    topcol;     /* top left corner of block's column */
COUNT    toprow;     /* top left corner of block's row */
COUNT    lstcol;     /* bottom right corner of block's column */
COUNT    lstrow;     /* bottom right corner of block's row */
```

DESCRIPTION
This function's purpose is to clear just a portion of the screen. Only the square block defined by the given parameters will be cleared. A block of spaces is written to stdio.

RETURN
Always returns a zero.

EXAMPLE
```
DT_CLRBK(10,12,20,14); /* clear block column 10, row 12 */
        /* to cloumn 20 row 14 */
```

REFERENCE NUMBER - 3F

NAME
DT_CMPAR- Compare elements of the Relate Structure. (DT_RELAT.C)

TYPE
Internal Structure Relationships

DECLARATION
```
COUNT    DT_CMPAR( cpp1, cpp2 , mode)
TEXT     *cpp1, *cpp2;
COUNT    mode;
```

DESCRIPTION
This function is the compare function called by the DT_RSORT routine to determine order.

Valid Modes- (defined in DT_TYPDF.H)

```
#define DTQSLTYP 1        /* Sort by left side type. */
#define DTQSLTAC 2        /* Sort by left side type and count. */
#define DTQSLTAA 3        /* Sort by left side type and alt sort. */
#define DTQSLSRT 4        /* Sort by left side alt sort field. */
#define DTQSRTYP 5        /* Sort by right side type. */
#define DTQSRTAC 6        /* Sort by right side type and count. */
#define DTQSRTAA 7        /* Sort by right side type and alt sort. */
#define DTQSRSRT 8        /* Sort by right side alt sort field. */
#define DTQSBAAC 9        /* Sort by left side alt and right side cnt */
#define DTQSBCAC 10    /* Sort by left side cnt and right side cnt */
#define DTQSKYBD 11    /* KEYBD sort-by terminal,key, no of gets */
#define DTQSHOOK 12    /* HOOKS sort-by spot (hook location) */
```

RETURN
Always returns a zero.

EXAMPLE
see the routine DT_RSORT.C

REFERENCE NUMBER - 43

NAME
DT_COMPI- Compile Time Incremental Structures. (DT_COMPI.C)

TYPE
Export/Import

DECLARATION
```
COUNT      DT_COMPI(fp,mode,dothese,totkeys,totsegs)
FILE       *fp;          /* file pointer for output */
COUNT      mode;         /* */
COUNT      dothese;      /* which incremental structures to create */
COUNT      totkeys, totsegs;
```

DESCRIPTION
Output c-tree incremental file definition structure to destination pointed to by given file pointer. This function is primarily used to dump the incremental file structure definitions from memory to a c source file. This file can then be included into a desired program at compile time.

Valid mode values:

0 - do all
1 - do top and middle
2 - do middle only
3 - do middle and bottom

Valid dothese values:

0 - do all
1 - do ISEGS
2 - do IIDXS
3 - do IFILS

RETURN
Always returns a zero.

EXAMPLE
DT_COMPI(fp,0,0,0,0);

SEE ALSO
DT_COMPL.c

REFERENCE NUMBER - 1F

NAME
DT_COMPL- Create initlized c source code structures for ability definitions. (DT_COMPL.C)

TYPE
Export/Import

DECLARATION
```
COUNT    DT_COMPL(filename)
TEXT     *filename;    /* file name to write definitions */
```

DESCRITPION
This function creates a source file with the initialized definition of the abilities currently defined in memory. It is normally used to create a compiled version of a program. Note the following: a d-tree script is parsed using the DT_PARSE function. Structures are initialized in memory with the parsed definition. Dumping these definitions to disk and including them at compile time would avoid the overhead of the parse. This function does this disk dump.

RETURN
NORMAL RETURN
zero returned for successful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x5401	ERR_5401	Could not open disk file.

EXAMPLE
```
if (DT_COMPL("MYSOURCE.c"))
        printf("Could not Write Compile specs\n");
```

REFERENCE NUMBER - 54

NAME
DT_CONST-Output a Constant Value to given screen location (DT_CONST.C)

TYPE
Screen Interface

DECLARATION
```
COUNT      DT_CONST(fp,ptr,tlcol,tlrow)
FILE              *fp;        /* pointer to destination */
DTTCONST          *ptr;       /* ptr to the def of desired constant */
COUNT             *tlcol;     /* top left base column number */
COUNT             *tlrow;     /* top left base row number */
```

DESCRIPTION
This function is primarily used to display a constant on the screen by passing stdout as the first parameter. An alternative file pointer could be passed. The constant definition contains the coordinates that are provided to DT_LOCAT function for positioning on the screen. If the file pointer is anything but stdout then DT_LOCPT is called to do the positioning as if writing to a text file. Optional top left row and column offsets can be passed that will be added to the base coordinates of the constant thus allowing constant repositioning via program control.

RETURN
Always returns a zero.

EXAMPLE
DT_CONST(fp,rptr-rptr,iptr-topcol,iptr-toprow);

SEE ALSO -
DT_IMGMV(); /* Image output with moving */

REFERENCE NUMBER - 20

NAME
DT_CPMEM- Allocate/Copy Ability Memory Block Definition Space. (DT_UTILY.C)

TYPE
Utility

DECLARATION
```
COUNT    DT_CPMEM(oldcnt,newcnt,global,size)
COUNT    *oldcnt;     /* number of units in old definition block */
COUNT    newcnt;      /* no of new units to add to definition block */
TEXT     **global;    /* pointer to old block of memory */
COUNT    size;        /* size of definition units */
```

DESCRIPTION
This function is used by most parsing routines to allocate the necessary memory space to store the parsed definition for an ability. If space has already been allocated for the current ability (designated by oldcnt0) then a new block of memory is allocated that is big enough to contain the existing definition plus the space for the new definition. The existing definition is copied to the new memory block and the original space is freed. The global pointer that was pointing to the existing definiton is set to point to the new memory block.

RETURN
NORMAL RETURN
A 0 is returned for successful completion.
ERROR RETURN
A 1 is returned if the new space could not be allocated.

EXAMPLE
```
if ( DT_CPMEM(&DTNIMAGE,1,
        (TEXT **) &DTGIMAGE,sizeof(DTTIMAGE)) )
                printf("Could not allocate space for IMAGE definiton");
```

REFERENCE NUMBER - 2C

NAME
DT_DELET- Delete Characters from a string. (DT_UTILY.C)

TYPE
Utility

DECLARATION

```
COUNT      DT_DELET(string,pos,no)
TEXT       *string;       /* pointer to string */
COUNT      pos;           /* starting position for delete */
COUNT      no;            /* number of characters to delete */
```

DESCRIPTION
This function is used to delete one or more characters from a string. Given the string pointer and the starting position (pos) in the string (0 .. up to .. strlen(string)-1) this function will remove the given number of characters (no) from the string.

RETURN
NORMAL RETURN
returns a zero for successful completion.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x3701	ERR_3701	Delete position beyond end of string.
0x3702	ERR_3702	Trying to delete too many characters.

EXAMPLE

```
char ray[64];

strcpy(ray,"1234567890");

if (DT_DELET(ray,1,20))
        printf("Trying to delete too many chacters");

if (!DT_DELET(ray,1,2))
        printf("string now contains '14567890'");
```

REFERENCE NUMBER - 37

NAME
DT_DFALT- Execute default logic for the given field and default type. (DT_DFALT.C)

TYPE
Data Management

DECLARATION
```
COUNT     DT_DFALT(fptr,type,tlcol,tlrow)
DTTFIELD  *fptr;      /* pointer to field definition */
COUNT     type;       /* type of default to look for */
COUNT     tlcol;      /* top left base column number for display*/
COUNT     tlrow;      /* top left base row number for display */
```

DESCRIPTION
This function looks to see if a field has a default definition for the given type. If it finds a match, then the default logic is executed.
Example: If the default key is hit when entering a field (FairCom uses the <TAB> key for the default key), this function is called. A pointer is passed to the active field and the TAB type of default. This function will look to see if there is a TAB definition for this field. If so, it initializes the field with the value from the default definition and displays the field.
Valid default types:
TAB - default when default key is hit.
INIT - default at initialization time.
DUPTAB - auto dup when auto dup key is hit.
DUPINIT - auto dup at initialization time.

RETURN

NORMAL RETURN
Number of default definitions found. If no defaults were defined for this field a zero is returned, otherwise the number of default definitions for this field is returned.
(NOTE: this is the total number of defaults defined for this field for all default types not just for type that was passed.)

ERROR RETURN

Value	Symbolic Constant	Explanation
0x7801	ERR_7801	Memory Allocation error on extract.

EXAMPLE

```
if (kbd= =DTKBAD)
        {
                DT_DFALT(fptr,DTDFTAB,iptr-topcol,iptr-toprow);
                kbd=DTKBCR;
        } /* DFALT */
```

REFERENCE NUMBER - 78

NAME
DT_DFIMG- Execute default logic for all fields associated with an image. (DT_DFALT.C)

TYPE
Data Management

DECLARATION
```
COUNT    DT_DFIMG(imageno)
COUNT    imageno;           /* Image number */
```

DESCRIPTION
This function will find all the INIT type of defaults for the fields associated with the given image. It will then execute the INIT default logic for all fields found. At this point in development this function is inefficient. We recommend using the DT_DFINI instead of this function to INIT all fields for an image. We provide this function at this time as an example of an extract that is based on another extract.

RETURN
NORMAL RETURN
Return a zero for successful completion.
ERROR RETURN VALUES

Value	Symbolic Constant	Explanation
0x9201	ERR_9201	Image number passed is not defined.
0x9202	ERR_9202	Could not allocate memory for extract.

EXAMPLE
```
if (DT_DFIMG(3))
        printf("Could not Initialize value on this image");
```

REFERENCE NUMBER - 92

NAME
DT_DFINI- Execute the INIT defaults within the given DEFAULTS definitions. (DT_DFINI.C)

TYPE
Data Management

DECLARATION
```
COUNT    DT_DFINI(dfaltno)
COUNT    dfaltno;      /* pointer to default definition */
```

DESCRIPTION
This function looks at all definitions within the given DEFAULTS for all INIT types. For all INIT types found, the associated fields will be initialized with the value from the default definition.

RETURN
NORMAL RETURN
Returns the number of defaults found.
(Note: this is the total number of defaults found within the given default definition, no just the INIT types.)
ERROR RETURN

Value	Symbolic Constant	Explanation
0x6D1	ERR_6D1	Invalid Default Number.
0x6D2	ERR_6D2	Memory Allocation error on extract.

EXAMPLE

This example's results:
1) cou_office will be initialized to "Reno Office"
2) cou_name will NOT be affected.
3) cou_date will be initialized to the sytem's date.

```
/* d-tree script syntax */
DEFAULTS(mydefaults)
/*      Symbol Name     Type of defaults     Defaults value */
        cou_office      INIT                 Reno Office
        cou_name        TAB                  Unknown
        cou_date        INIT                 SYSDATE

/* c source file */
myfunct()
{
        int mydefaults;

        mydefaults=DT_INAME("mydefaults");
        DF_DFINI(mydefaults); /* execute initialization */
        return(0);
} /* end example function */
```

REFERENCE NUMBER - 6D

NAME
DT_DLREC- Delete a record from a c-tree data file. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_DLREC(datno)
COUNT     datno;/* data file number */
```

DESCRIPTION
This function will delete the current isam record from a c-tree file. It first determines if the file is variable length or not and calls either c-tree's DELVREC or DELREC.

RETURN
NORMAL RETURN
Returns a zero for successful delete.
ERROR RETURN
Returns the c-tree DELREC or DELVREC error if delete fails.

EXAMPLE
```
if (err=DT_DLREC(datno))
{
        sprintf(target,"Unable to delete record c-tree error = %d",err);
        DT_DOMSG(target); getchar();
} /* end if delete error */
```

SEE ALSO
c-tree's DELREC and DELVREC.

REFERENCE NUMBER - 2D

NAME
DT_DODBK- Check if DODA entry is Blank (or zero). Check to see if the field address pointed to by the given DODA entry contains a value. (DT_FIELD.C)

TYPE
DODA Management/Internal Structure Relationships

DECLARATION
```
COUNT     DT_DODBK(fdoda,isvalue,fvalue)
DATOBJ    *fdoda;      /* pointer to DATOBJ entry */
LONG      *ivalue;     /* pointer to LONG */
double    *fvalue;     /* pointer to double */
```

DESCRIPTION
Given a DATOBJ pointer, check to see if the associated field address contains a value. Alpha fields are check for characters and numeric fields are checked for zero. If a value exists and is alpha, the value is pointed to by the doda. If it is a long integer, this function returns a pointer to the value as the second parameter. Likewise, if the value is floating point, a pointer to the value is returned as the third parameter.

RETURN
Returns a zero if field has a value.
Returns a 1 if field is blank or zero.

EXAMPLE
```
if (DT_DODBK(dodaptr,longptr,floatptr))
        printf("value is zero");
```

REFERENCE NUMBER - 7B

NAME
DT_DODTS- Create default d-tree script. (DT_DODTS.C)

TYPE
Export/Import

DECLARATION

```
COUNT     DT_DODTS(script,dtstyp,flag)
TEXT      *script;      /* Source script to aid creation */
COUNT     dtstyp;       /* type of pgm script to create */
COUNT     flag;         /* where to get field text flag */
```

DESCRIPTION
This function creates a default d-tree script. It assumes that the there is a DODA initialized with the fields to be used for the script.

RETURN
NORMAL RETURN
return 0 for successful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x8901	ERR_8901	Could not open base script.
0x8902	ERR_8902	Could not write temp dtree script.
0x8903	ERR_8903	Could not open user script.

EXAMPLE

```
/* create d-tree script section */
if (DT_DODTS(DTDTSSTD,script))
        printf("Could not Create d-tree script");
```

REFERENCE NUMBER - 89

NAME
DT_DOINT- Initialize d-tree record buffers. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_DOINT(datno)
COUNT     datno;
```

DESCRIPTION
This function initializes the record buffers associated with the given data file number. d-tree allocates three record buffers for record maintenance. All three buffers are filled with NULLs.

RETURN
Always returns a zero

EXAMPLE
DT_DOINT(1); /* initializes file number 1 buffers */

SEE ALSO
DT_INBUF(); /* initialized memory buffer */

REFERENCE NUMBER - 34

NAME
DT_DOMSG- Display a message on the screen on the default message line. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
```
COUNT       DT_DOMSG(msg)
TEXT        *msg;          /* pointer to message */
```

DESCRIPTION
Displays a message on the default message line. The default message line is defined in DT_TYPDF.H.

#define DT_MSGLN 24 /* Error message default line */

The message text is centered on the message line. This function is used by d-tree to display error messages from the edits functions.

RETURN
Always returns a zero.

EXAMPLE
```
if (err=DT_ADREC(datno))
{
        sprintf(target,"Unable to Add record c-tree error = %d",err);
        DT_DOMSG(target); getchar();
} /* end if add error */
```

REFERENCE NUMBER - 91

NAME
DT_DOPAD- Pad fixed length fields in given data file record buffer. (DT_CTREE.C)

TYPE
File I/O

DECLARATION

```
COUNT     DT_DOPAD(datno)
COUNT     datno;        /* data file number */
```

DESCRIPTION
This function is used to pad the fixed length fields in a given data file's record buffer with the PADDING character defined in c-tree. This function is normally called before a record ADD to ensure key values are consistent.

RETURN
This function always returns a zero.

EXAMPLE
DT_DOPAD(1); /* pad fields for file number 1 */

REFERENCE NUMBER - 35

NAME
DT_DORTS- Create default r-tree script. (DT_DORTS.C)

TYPE
Export/Import

DECLARATION
```
COUNT    DT_DORTS(script,myfile,openmode,flag)
TEXT     *script;      /* script name to create */
COUNT    myfile;/* file number for SEARCH default */
COUNT    openmode;  /* open file mode */
COUNT    flag;         /* constant text flag */
```

DESCRIPTION
This function creates a default r-tree script. It scans the relate structure and writes an entry into the r-tree script for each FIELD type found.

RETURN
NORMAL RETURN
return 0 for successful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x9001	ERR_9001	Could not write new r-tree script.
0x9002	ERR_9002	Could not open d-tree script.
0x9003	ERR_9003	Could not find master screen.

EXAMPLE
```
if (DT_DORTS("MY.RTS",1))
        printf("Could not create r-tree script");
```

REFERENCE NUMBER - 90

NAME
DT_EDATE- Edit Date (DT_EDITS.C)

TYPE
Data Management

DECLARATION

```
COUNT     DT_EDATE(dodaptr,edttyp)
DATOBJ    *dodaptr;     /* field to edit */
COUNT     edttyp;       /* type of date edit */
```

DESCRIPTION
This function validates the specified field as a valid entry of the date type passed. Valid date types are:
DTETDAT1 - DATE edit.-MMDDYY.
DTETDAT2 - DATE edit.-MMYY.
DTETDAT3 - DATE edit.-MMDD.

Additional date types may be added by the user. (See DT_TYPDF.H for edit type #defines and DT_EDITS for source file)

RETURN
Any non zero value indicates edit failed.
A zero value means that the field passed the edit.

EXAMPLE

```
error=DT_EDATE(((DTTFIELD *)(rptr-lptr))-fdoda,
        ((DTTEDITS *)(rptr-rptr))-edttyp);
```

SEE ALSO -
DT_EDITS(); /* primary edit function */

REFERENCE NUMBER - 2E

NAME
DT_EDITS- Execute edit logic defined for a certain field or all fields within one EDITS section. (DT_EDITS.C)

TYPE
Data Management

DECLARATION
```
COUNT      DT_EDITS(fptr,editno)
DTTFIELD   *fptr;      /* field pointer to edit */
COUNT      editno;     /* EDITS number for edit definitions */
```

DESCRIPTION
This function can be used in two ways:
1) If a field pointer is passed, execute all edits defined for this field. This mode is called upon input of each field.
2) If an EDITS number is passed, all edits defined in this EDITS section are executed. This mode is usually called to edit all fields at once, prior to disk update.
If an error is detected then the error message is displayed via the DT_DOMSG function and the error id is returned.

RETURN
NORMAL RETURN
If an edit passes then zero is returned, otherwise, if an edit fails, it's edit id is returned.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x6701	ERR_6701	Memory Allocation error on extract.

EXAMPLE
```
if (DT_EDITS((DTTFIELD *)0,1))
        printf("Edit for EDITS(1) failed: DO NOT POST DATA");
```

SEE ALSO-
```
DT_DOMSG();     /* display message /
DT_EMAND();     /* MANDATORY edit. */
DT_EFILL();     /* MANDATORY fill edit. */
DT_EDATE();     /* DATE edit. */
DT_ETABL();     /* TABLE edit. */
DT_EDUPK();     /* Duplicate key edit. */
DT_EVALD();     /* VALIDATE edit. */
DT_ESFLH();     /* Subfile Hash edit. */
```

REFERENCE NUMBER - 67

NAME
DT_EDUPK- Duplicate Key Edit (DT_EDITS.C)

TYPE
Data Management

DECLARATION

```
COUNT     DT_EDUPK(dodaptr,txtptr)
DATOBJ    *dodaptr;    /* Field Pointer */
TEXT      *txtptr;     /* key number for edit */
```

DESCRIPTION
This function performs c-tree's TFRMKEY on the record buffer associated with the given field pointer, using the key number contained in txtptr. The target from TFRMKEY is used to see if an entry already exits in the index for this value.

RETURN
A non zero value indicates duplicate key found.
Zero means field passed edit.

EXAMPLE
This edit function is called from DT_EDITS. See DT_EDITS.c

SEE ALSO -
DT_EDITS(); /* primary edit function */

REFERENCE NUMBER - 2F

NAME
DT_EFILL- Mandatory Fill Edit (DT_EDITS.C)

TYPE
Data Management

DECLARATION
```
COUNT     DT_EFILL(dodaptr,length)
DATOBJ    *dodaptr;     /* pointer to field */
COUNT     length;       /* length of field */
```

DESCRIPTION
This function is called to ensure every character in the given field has a character.

RETURN
Returns a value is field does not have a character in every byte.
Returns a zero is all bytes have character.

EXAMPLE
This edit function is called from DT_EDITS. See DT_EDITS.c

SEE ALSO -
DT_EDITS(); /* primary edit function */

REFERENCE NUMBER - 3D

NAME
DT_EMAND- Mandatory Field Edit (DT_EDITS.C)

TYPE
Data Management

DECLARATION
```
COUNT       DT_EMAND(dodaptr)
DATOBJ      *dodaptr;    /* field pointer */
```

DESCRIPTION
This function checks to ensure that data has been entered into the given field.

RETURN
Returns a value if field does not have data.
Returns a zero if field has data.

EXAMPLE
This edit function is called from DT_EDITS. See DT_EDITS.C

SEE ALSO -
DT_EDITS(); /* primary edit function */

REFERENCE NUMBER - 3A

NAME
DT_EQREC- Get a data record with key value equal to target value. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT   DT_EQREC(keyno,target)
COUNT   keyno;       /* key number for get */
TEXT    *target;     /* target to use for get */
```

DESCRIPTION
This function performs the following steps:
a) c-tree's EQLREC to get the record. if error on EQLREC return error.
b) Determines if the file is variable length or not. if variable length it then un-packs
the record into the maintenance buffer otherwise it simply copies the read record into the maintenance buffer.
c) if UNIFRMAT is defined, the uniformat logic is executed.
d) DT_UNPAD is called to unpack fixed length fields.

RETURN
NORMAL RETURN
Returns a zero for successful get.
ERROR RETURN
Returns the c-tree EQLREC error if failed.

EXAMPLE
```
if (!(DT_EQREC(keyno,target)))
        printf("Got a Hit");
```

REFERENCE NUMBER - 4A

NAME
DT_ETABL- Table Edit Function. Ensure that field value is in a given TABLE of values. (DT_EDITS.C

TYPE
Data Management

DECLARATION

```
COUNT     DT_ETABL(dodaptr,txtptr)
DATOBJ    *dodaptr;    /* field to check */
TEXT      *txtptr;     /* list of table entries */
```

DESCRIPTION
This function scans the list of table elements to ensure that the value of the given field is contained in the table. A good example of a table edit is a Y/N validation where you only want the user to be able to key a Y or a N.This edit function is called from DT_EDITS. See DT_EDITS.C

RETURN
A nonzero value is returned if field's value is not found in the table.

A zero is returned if the value is in the table.

EXAMPLE
/* d-tree script */
EDITS(master)

Must Enter a Y or a N cou_cod TABLE Y N

SEE ALSO -
DT_EDITS(); /* primary edit function */

REFERENCE NUMBER - 3E

NAME
DT_EVALD- Validation edit. Edit the value of a field as key to another file. (DT_EDITS.C)

TYPE
Data Management

DECLARATION
```
COUNT      DT_EVALD(fptr,txtptr)
DTTFIELD   *fptr;         /* field to edit */
TEXT       *txtptr;       /* validation definition string */
```

DESCRIPTION
This function validates the given field as a key value in the given key. This function is used to edit input such as customer number or account number, or any value that can be edited against a unique key. (Customer master Customer number key or Account master account key). A SCANN may be defined to allow a lookup into the associated file. A MAPIT may be defined if a select was done on the scann to map data back to the record being maintained. This edit function is called from DT_EDITS. See DT_EDITS.C

RETURN

NORMAL RETURN
Returns a zero if a match is found.

ERROR RETURN
Returns a non-zero value if no index match is found.

EXAMPLE
```
EDITS(master)
Invalid Code cou_cod VALIDATE ex2idx cmap cscann prefix
Index to validate.................................. ^
Map for selection.......................................... ^
Scan for lookup....................................................... ^
Prefix for scann................................................................ ^
```

SEE ALSO -
```
DT_EDITS();      /* primary edit function */
DT_MAPIT();      /* Map data */
DT_SCANN();      /* Scann Data file */
```

REFERENCE NUMBER - 3C

NAME
DT_EVALU- Evaluate a postfix expression. (DT_EVALU.C)

Note: This function will be supported in commercial version. Not running in beta version.

TYPE
Data Management

DECLARATION

```
COUNT     DT_EVALU(in,ivalue,fvalue)
TEXT      *in;          /* expression pointer */
LONG      *ivalue;      /* ptr to integer value */
double    *fvalue;      /* ptr to floating pt value */
```

DESCRIPTION
Using a stack, DT_EVALU evaluates the postfix expression and performs the appropriate calculations giving the desired result. It is used by the DT_CALCS to perform the calculations defined in the d-tree script.

RETURN
NORMAL RETURN
Returns zero if calculation was successful.

ERROR RETURN
Returns 1 if error occurred.

EXAMPLE

```
if (DT_EVALU(string))
        printf("unable to evaluate expression");
```

SEE ALSO
DT_CALCS(); /* perform defined calculation */
DT_PSTFX(); /* convert a infix expression to postfix */

REFERENCE NUMBER - 6B

NAME
DT_FLDIN- Control input for given field (DT_FIELD.C)

TYPE
Screen Interface

DECLARATION

```
COUNT       DT_FLDIN(ptr,tlcol,tlrow)
DTTFIELD    *ptr;           /* field to input */
COUNT       tlcol;          /* top left base column number */
COUNT       tlrow;          /* top left base row number */
```

DESCRIPTION
To manage input for a specified field. This function will accept input for a given field, taking into account attributes, screen location, and field type.

RETURN
NORMAL RETURN
Returns last key hit during maintenance.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x2501	ERR_2501	Input Longer than max field length DT_FLDLN. (see DT_TYPDF.H)
0x0031	DTKBCR	Specified field is protected.

EXAMPLE

```
/* INPUT */
if ( (kbd = DT_FLDIN(fptr,iptr-topcol,iptr-toprow)) = = DTKBESC )
        break;
```

SEE ALSO -

```
DT_FLDTX();     /* convert text to field */
DT_INPUT();     /* input function */
DT_FLDOT();     /* field out */
DT_LOCAT();     /* screen locate */
DT_SCSEQ();     /* screen special sequence */
DT_INBUF();     /* initilize buffer to nulls */
```

REFERENCE NUMBER - 25

NAME
DT_FLDLO- Low level field output routine (DT_FIELD.C)

TYPE
Screen Interface

DECLARATION

```
COUNT      DT_FLDLO(fp,ptr,lstno,spcatr)
FILE       *fp;          /* file pointer for output */
DTTFIELD   *ptr;         /* field pointer */
COUNT      *lstno;       /* number of characters output */
COUNT      spcatr;       /* table number to do lookup */
```

DESCRIPTION
This function outputs a field to the given file pointer (stdout). It is called from the DT_FLDOT function to output a field's contents to the screen.

RETURN
If there is no data in the field then a 1 is returned.
If field contains data then a 0 is returned.
If field is alpha then the number of characters that were output isreturned in lstno.

EXAMPLE
blank=DT_FLDLO(stdout,ptr,&x,spcatr);

SEE ALSO -
DT_FLDOT(); /* output a field */

REFERENCE NUMBER - 7A

NAME
DT_FLDNM- Validate a token as a valid field symbol (DT_FIELD.C)

TYPE
Table Validation

DECLARATION
```
DTTFIELD    *DT_FLDNM(token)
TEXT        *token;         /* token to validate */
```

DESCRIPTION
This function validates a token as a valid FIELD symbol name in the DODA. If a match is found then a pointer to the FIELD definition is returned. If no match is found a zero is returned.

RETURN
NORMAL RETURN
pointer of DTTFIELD type to field found.
ERROR RETURN
0 for no such field.

EXAMPLE
```
if (!DT_FLDNM("name"))
        printf("Not a valid field symbol");
```

REFERENCE NUMBER - 70

NAME
DT_FLDOT- Output the contents of a field (DT_FIELD.C)

TYPE
Screen Interface

DECLARATION

```
COUNT      DT_FLDOT(fp,ptr,tlcol,tlrow)
FILE       *fp;          /* file for output */
DTTFIELD   *ptr;         /* field pointer to output */
COUNT      tlcol;        /* top left base column number */
COUNT      tlrow;        /* top left base row number */
```

DESCRIPTION
This function is used to display the contents of the field. If the field is blank or zero, then underscores ("________") will be displayed.

RETURN
Always returns a zero.

EXAMPLE
DT_FLDOT(stdio,fldptr,iptr-topcol,iptr-toprow);

REFERENCE NUMBER - 21

NAME
DT_FRAME- Draw a frame on the screen. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
```
COUNT     DT_FRAME(fp,tp,tx,ty,bx,by)
FILE      *fp;            /* file pointer for output */
TEXT      *tp;            /* pointer to text to be included at top line */
COUNT     tx,ty,bx,by;    /*top left col,row; bottom right col,row*/
```

DESCRIPTION
Draw a frame to the given file pointer(stdout).

RETURN
always returns a zero.

EXAMPLE
DT_FRAME(01,01,23,23);

REFERENCE NUMBER - 4C

NAME
DT_FREEE- Free Ability Memory Blocks. (DT_FREEE.C)

TYPE
DODA Management/Internal Structure Relationships

DECLARATION
COUNT DT_FREEE()

DESCRIPTION
This function will free the allocated memory for d-tree abilities that where allocated from parsing routines or that came from the Ability dictionary. Primarily used in catalog master program.

RETURN
Always returns a zero.

EXAMPLE
DT_FREEE();

REFERENCE NUMBER - 2A

NAME
DT_FSREC- d-tree get first record. Get the first record in a file. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT    DT_FSREC(filno)
COUNT    filno;        /* data or index file number */
```

DESCRIPTION
This function performs the following steps.
a) calls c-tree's FRSREC to get the first record. if error on FRSREC, return error.
b) determines if the file is variable length or not. If variable length, it then un-packs
the record into the maintenance buffer otherwise it simply copies the read record into the maintenance buffer.
c) if UNIFRMAT is defined, the uniformat logic is executed.
d) DT_UNPAD is called to unpack fixed length fields.

RETURN
NORMAL RETURN
Returns a zero for successful get.
ERROR RETURN
Returns the c-tree FRSREC error if failed.

EXAMPLE
```
if (!(DT_FSREC(keyno)))
        printf("Got first record in key order");
```

REFERENCE NUMBER - 99

NAME
DT_FUNCT- Validate a token as a valid user defined function. (DT_FUNCT.C)

TYPE
Table Validation

DECLARATION
```
DT_FPTR     DT_FUNCT(token)
TEXT        *token;        /* token to validate */
```

DESCRIPTION
This function is used to validate a token as a user defined function in the structure dt_user. If a token is the name of a user defined function, a pointer to that function is returned.

```
/***************************************/
/*    Valid User defined special functions    */
        DTTFUNCT dt_user[] = {
                { "My_Function" , myfunc },
                 { "", DT_NULFP }    /* terminaton Indicator */
        };
/***************************************/
```

RETURN
Either a zero for no match or a pointer to the associated function.

EXAMPLE
```
thefunction()
{
        DT_FPTR funcptr;

        /* USER FUNCTION */
        funcptr=DT_FUNCT("My_Function");
        if (funcptr!=DT_NULFP)
                (*funcptr)(); /* call function */

        return(0);
} /* end sample function */
```

REFERENCE NUMBER - 69

NAME
DT_GENRL- Validate Token in a General Table. (DT_GENRL.C)

TYPE
Table Validation

DECLARATION

```
DTTGENRL  *DT_GENRL(base,token,addit)
DTTGENRL  *base;      /* base address of table */
TEXT      *token;     /* token to validate */
COUNT     addit;      /* add to table if not in table flag */
```

DESCRIPTION
This function is used to do a lookup into tables defined as DTTGENRL d-tree uses this lookup capability for a variety of validation tables.Note typdef DTTGENRL:

```
    /* GENERAL valid token structure typedef */
    typedef struct {
        TEXT  *string;      /* ptr to string for token */
        COUNT refnum;       /* reference number */
        COUNT type;         /* reference type */
    } DTTGENRL;
```

RETURN
NORMAL RETURN
Returns the pointer to the table if a match was found.
ERROR RETURN
Returns a zero if no match was found.

Value	Symbolic Constant	Explanation
0x1801	ERR_1801	DT_GENRL called with invalid table.
0x1802	ERR_1802	DT_GENRL could not allocate space for symbolic names.
0x1803	ERR_1803	DT_GENRL could not allocate space for symbolic name text.

EXAMPLE

```
/***************************************/
/* Valid EDITS types symbols */
DTTGENRL dt_genedt[] = {
        {"MANDATORY",DTETMAND },        /* mandatory entry */
        {"MAND_FILL",DTETFILL },        /* mandatory fill */
        {"DATE_MMDDYY",DTETDAT1 },      /* date edit-MMDDYY */
        {"DATE_MMYY",DTETDAT2 },        /* date edit-MMYY */
        {"DATE_MMDD",DTETDAT3 },        /* date edit-MMDD */
        {"TABLE",DTETTABL },            /* validate against a table */
        {"DUPKEY",DTETDUPK },           /* check for duplicate keys */
        {"VALIDATE",DTETVALD },         /* validate against key */
        {"",-1}                         /* terminaton indicator */
};
/***************************************/
mytest()
{
        DTTGENRL *gptr;

        if ((gptr = DT_GENRL(dt_genedt,"DUPKEY",0)))
                printf("reference number = %d\n",gptr-refnum);
}
```

REFERENCE NUMBER - 18

NAME
DT_HELPP- execute help function. (DT_HELPP.C)

TYPE
Screen Interface

DECLARATION
```
COUNT     DT_HELPP(ptr,type)
TEXT      *ptr;   /* pointer to entity for help */
COUNT     type;   /* type of help */
```

DESCRIPTION
This function is used to display help text for the given ability entry. It will either display the help text defined in the d-tree script or it will access the help text file with the defined token to access the help. Help is either displayed on the message line or in a subfile display area, depending upon the d-tree script definition.

RETURN
NORMAL RETURN
Returns zero for successful completion
ERROR RETURN

Value	Symbolic Constant	Explanation
0x851	ERR_851	Could not initialize help text subfile.
0x852	ERR_852	Subfile defined for help text not found.

EXAMPLE
see DT_IMGIN in DT_IMAGE.c

```
if (kbd = =DTKBHELP) /* help text */
        {
                DT_HELPP( (TEXT *)fptr,DTKFIELD);
                error=1;
                continue;
        }
```

REFERENCE NUMBER - 85

NAME
DT_IFILS- Open/Create Incremental Files (DT_IFILS.C)

TYPE
File I/O

DECLARATION

```
COUNT     DT_IFILS(bufs,extra,sect)
COUNT     bufs;  /* number of index file buffers */
COUNT     extra; /* number of extra files not opened by DT_IFILS*/
COUNT     sect;  /* number of node sectors */
```

DESCRIPTION
This function will first count all data file and index files defined in the incremental structures in order to execute c-tree's INTISAM. It will then Open (or Create if file not found) each file defined using c-tree's OPNIFIL (or CREIFIL). If a file is found to be corrupt at open, automatic rebuilding of the file will be executed if the #define DTRBLIFIL is set in DT_DEFIN.H

RETURN
NORMAL RETURN
Returns a zero for successful completion.
ERROR RETURN
If function fails, returns c-tree's error from INTISAM, OPNIFIL or CREIFIL

EXAMPLE

```
if (err=DT_IFILS(10,0,4))
        printf("\nError Occured During Open IFIL=%d\n",err);
```

REFERENCE NUMBER - 81

NAME
DT_IMAGE - Display and Input from an Image (DT_IMAGE.C)

TYPE
Screen Interface

DECLARATION

```
COUNT     DT_IMAGE(imageno)
COUNT     imageno;            /* image number */
```

DESCRIPTION
This function will display the specified IMAGE by calling the DT_IMGOT function. It then calls the DT_IMGIN function to accept input for variable fields on the IMAGE.

RETURN
NORMAL RETURNS
This function returns that last key detected from the keyboard.
ERROR RETURNS
via IMGOT:

Value	Symbolic Constant	Explanation
0x1901	ERR_1901	invalid image number.
0x1902	ERR_1902	image has no fields.
0x1903	ERR_1903	could not open print file.
0x1904	ERR_1904	can't Use TEMPP type already in use.
0x1905	ERR_1905	can't allocate TEMPP typespace for print screen.

via IMGIN:

Value	Symbolic Constant	Explanation
0x2601	ERR_2601	Invalid image number.
0x2602	ERR_2602	image has no fields.

EXAMPLE

```
if ((DT_IMAGE(menu)) = =DTKBESC)
        printf("Escape Was Hit");
```

REFERENCE NUMBER - 27

NAME -
DT_IMGAL - Display and Input a group of IMAGES (DT_IMAGE.C)

TYPE -
Screen Interface

DECLARATION -

```
COUNT    DT_IMGAL(imageno,begimg,endimg)
COUNT    imageno;          /* starting image number */
COUNT    begimg;           /* lowest image number */
COUNT    endimg;           /* highest image number */
```

DESCRIPTION -
This function can be called for your c program to display and input more than one image at a time. The first image to be displayed is passed to this function by imageno. This function assumes that the images to be displayed are numbered sequentially. As the user pages up and down the previous or next image will be displayed. The beginning and ending set the upper and lower bounds respectively for the image numbers. Note that this function is simply a loop setting imageno and calling DT_IMAGE().

RETURN
Returns the last keystroke from the keyboard.

EXAMPLE

```
if ((DT_IMGAL(1,1,10)) = =DTKBESC) /* maintain 10 screens */
        printf("Escape Hit");
```

REFERENCE NUMBER - 28

NAME
DT_IMGIN - Input an Image (DT_IMAGE.C)

TYPE
Screen Interface

DECLARATION
```
COUNT      DT_IMGIN(imageno,curfld,curptr)
COUNT      imageno;     /* image number */
COUNT      *curfld;     /* current field number */
DTTFIELD   **curptr;    /* current field pointer */
```

DESCRIPTION
This function manages the input of variable fields pertaining to an IMAGE using the relate structure to access field definitions related to this IMAGE. For each related field found, a DT_FLDIN (field in) function call is made. The field input order is controlled by this function, responding to input keys (UP, DOWN, etc.) Based on the last keystroke entered, other special functions are also called from this function; such as:

DT_DFALT - if default key was hit, execute default logic.
DT_HELPP - execute help function if help key was hit.
DT_EDITS - if not special function key was hit, edit the field.
DT_IMGOT - Image out is called w/output redirected if the print screen key hit.

RETURN
NORMAL RETURN
Returns the last keystroke hit from keyboard.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x2601	ERR_2601	DT_IMGIN-invalid image number.
0x2602	ERR_2602	DT_IMGIN-image has no fields.

EXAMPLE

```
if ( (keystroke=DT_IMGIN(start,&kbd,&notused)) == DTKBESC)
        return(-1);
switch (keystroke)
        {
        case DTKBUP: doifup();          /* up key */
                break;
        case DTKBHM: doifhome();        /* home key */
                break;
        } /* end switch */
```

SEE ALSO -

DT_HELPP - help function.
DT_IMGOT - output an IMAGE.
DT_DFALT - execute default logic.
DT_FLDIN - input a field.
DT_EDITS - edit a field.

REFERENCE NUMBER - 26

NAME
DT_IMGLG - re-display IMAGE's from the image log (DT_IMAGE.C)

TYPE
Screen Interface

DECLARATION
```
COUNT      DT_IMGLG(level);
COUNT      level;          /* log level to display */
```

DESCRIPTION
This function is used to re-display an image that has been previously written to the screen. When the DT_IMGOT (image out) function is called, it is optionally logged in the IMAGE LOG. When another screen is written it may overlay the current screen. The DT_IMGLG function is used to re-display the IMAGE that was overlayed. The image log is a two dimensional array. A clear screen will provoke a new level in the log. The level number parameter indicates which level to display.

RETURN
Always returns a zero.

EXAMPLE
```
if (scroverlay)
DT_IMGLG(1); /* redisplay level 1 */
```

SEE ALSO -
DT_IMGIN - IMAGE In function.

REFERENCE NUMBER - 1A

NAME
DT_IMGMV - Display/Input an Image allowing new screen coordinates. Screen will move to the new coordinates. (DT_IMAGE.C)

TYPE
Screen Interface

DECLARATION
```
COUNT       DT_IMGMV(imageno)
COUNT       imageno;      /* image number */
```

DESCRIPTION
This function is the same as the DT_IMAGE function except using the function keys F1 thru F4 causes the screen to change it's base coordinates and be re-displayed. This causes the IMAGE to move around the screen under control of the user.

RETURN
NORMAL RETURN
Returns the last keystoke from the keyboard.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x4701	ERR_4701	invalid image number.

via IMGOT:

Value	Symbolic Constant	Explanation
0x1901	ERR_1901	invalid image number.
0x1902	ERR_1902	image has no fields.
0x1903	ERR_1903	could not open print file.
0x1904	ERR_1904	tried to Use TEMPP type that is already in use.
0x1905	ERR_1905	could not allocate TEMPP type space for print screen.

via IMGIN:

Value	Symbolic Constant	Explanation
0x2601	ERR_2601	invalid image number.
0x2602	ERR_2602	image has no fields.

EXAMPLE
```
if ((DT_IMGMV(menu)) = =DTKBESC)
        printf("Escape Hit");
```

SEE ALSO -
DT_IMAGE - Output/Input an Image.

REFERENCE NUMBER - 47

NAME
DT_IMGOT- Ouput an IMAGE (DT_IMAGE.C)

TYPE
Screen Interface

DECLARATION

```
COUNT     DT_IMGOT(imageno,mode,special,speccnt)
COUNT     imageno;    /* image number */
COUNT     mode;       /* display mode */
COUNT     special;    /* image logging type */
COUNT     speccnt;    /* special image logging parameter */
```

DESCRIPTION
This function is used to display an image to the screen. It accesses all variable and constant fields related to the IMAGE via the relate structure. As it finds a related field, it either does a DT_CONST (display constant field) or a DT_FLDOT (variable field out) depending on the variable type.
IMAGE display modes:
DTIMGALL - display both constant and variables.
DTIMGCON - display constants only.
DTIMGVAR - display variables only.
DTIMGPWP - print screen (write & print).
DTIMGPWO - print screen (write only).
DTIMGPAP - print screen (append and print).
DTIMGPAO - print screen (append only).
IMAGE logging types:
DTIMGREG - nothing special for image out.
DTIMGLOG - displaying from log-do not log.
DTIMGSFL - image being displayed is subfile.
DTIMGNOL - No Log-do not log image.
Special Image Logging Parameter:
Used for additional information needed to re-display the image from the image log. Example: when a subfile is displayed, the special parameter is the subfile number.

RETURN

NORMAL RETURN

Returns a zero for successful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x1901	ERR_1901	invalid image number.
0x1902	ERR_1902	image has no fields.
0x1903	ERR_1903	could not open print file.
0x1904	ERR_1904	tried to Use TEMPP type that is already in use.
0x1905	ERR_1905	could not allocate TEMPP type space for print screen.

EXAMPLE

```
if ( (err=DT_IMGOT(muscreen,DTIMGALL,DTIMGREG,DTIMGREG)) )
        printf("Could not display screen error=%x\n",err);
```

SEE ALSO -

DT_CONST - output a constant field.

DT_FLDOT - output a variable field.

REFERENCE NUMBER - 19

NAME
DT_INAME- return the IMAGE number for the given string (DT_IMAGE.C)

TYPE
Table Validation

DECLARATION
```
COUNT   DT_INAME(name)
TEXT     *name;     /* image name */
```

DESCRIPTION
Given an image name, return the associated image number. This function simply does a lookup in the DT_GENRL table that contains the valid image names. If a match is found, the associated image number is returned, otherwise a zero is returned.

RETURN
NORMAL RETURN
Image number associated with string.
Zero for no match found.

EXAMPLE
```
if(!(menu=DT_INAME("mymenu")))
        printf("mymenu was not defined in d-tree script");
```

REFERENCE NUMBER - 53

NAME
DT_INBUF- Initialize a buffer. (KBDT_UTILY.C)

TYPE
Utility

DECLARATION
```
COUNT    DT_INBUF(dp,n)    /* initialize a buffer with nulls */
TEXT     *dp;
COUNT    n;
```

DESCRIPTION
The specified region (string) of memory is set to nulls.

RETURN
Always returns a zero.

EXAMPLE
```
TEXT name[32];

DT_INBUF(name,32); /* set name field to nulls */
```

REFERENCE NUMBER - 36

NAME
DT_INPUT-low level field input routine. (DT_INPUT.C)

TYPE
Screen Interface

DECLARATION
```
COUNT      DT_INPUT(field,col,row,len,maxlen,inpatr,outatr,mask)
TEXT       *field;        /* input buffer */
COUNT      col;           /* column on screen */
COUNT      row;           /* row on screen */
COUNT      len;           /* screen length to input */
COUNT      maxlen;        /* max length to input */
COUNT      inpatr;        /* input arribute */
COUNT      *outatr;       /* pointer to output attributes array */
TEXT       *mask;         /* input mask for each char edit */
```

DESCRIPTION
This function is used to maintain input for the specified field. It will position the cursor at the given coordinates, take into account input and output attributes, control input based on given lengths, allowing the contents of the variable field to be modified.

RETURN
Returns the last keystroke from the keyboard.

EXAMPLE
```
TEXT name[32];

exitkey=DT_INPUT(name,        /* field */
                  12,         /* column */
                  10,         /* row */
                  30,         /* screen length */
                  30,         /* max length */
                   0,         /* input attribute */
                   0);        /* output attribute */
```

REFERENCE NUMBER - 4D

NAME
DT_KEYBD- initialize screen and keyboard definitions from termcap file. (DT_KEYBD.C)

TYPE
Export/Import

DECLARATION

```
COUNT    DT_KEYBD(termcap,terminal)
TEXT     *termcap;          /* termcap file name */
TEXT     terminal;          /* terminal name */
```

DESCRIPTION
This function initializes the screen and keyboard definitions for the d-tree keyboard and screen routines. This function will read the specified termcap file name and look for the terminal name provided. When it finds the proper terminal definition, DTPKEYBD is called to parse in the definition.

RETURN
NORMAL RETURN
Returns a zero for successful completion.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x7201	ERR_7201	Could not open termcap file.
0x7202	ERR_7202	Terminal not found in termcap file.
0x7203	ERR_7203	Could not allocate temp parsing space.
0x7101	ERR_7101	Could not allocate space for key seq.
0x7102	ERR_7102	Syntax error in termcap definition.
0x7103	ERR_7103	DT_MXSEQ not large enough in DT_TYPDF.H.

EXAMPLE

```
if (err=DT_KEYBD(TERMCAP,getenv("TERM")))
{
        printf("\nError Occurred During TERMCAP Parse Error=%x\n",err);
        exit(1);
}
```

SEE ALSO -
DTPKEYBD - Parse KEYBD definition.

REFERENCE NUMBER - 72

NAME
DT_KEYNM- Validate token as a valid key symbolic name. (DT_PARSE.C)

TYPE
Table Validation

DECLARATION
```
COUNT     DT_KEYNM(token)
TEXT      *token;        /* token to validate */
```

DESCRIPTION
This function checks to see if the token that was passed is a key symbolic name defined in the ISAM definition.

RETURN
If a match is found, the associated key number is returned; otherwise a zero is returned indicating that the token is not a key symbolic name.

EXAMPLE
```
if (OPNISAM(EXAMPLE.P)) /* must open the isam for DT_KEYNM to work */
{
        printf("\n\nCould not open isam. Error codes %d %d",isam_err,isam_fil);
        exit(0);
}

if (keyno=DT_KEYNM(token))
        printf("Symbol %s refers to key number %d\n",keyno);
else
        printf("Symbol %s IS NOT in valid key symbol name\n",token);

if (CLISAM())
{
        printf("\n\nCould not close isam. Error codes %d %d",isam_err,isam_fil);
}
```

REFERENCE NUMBER - 42

NAME
DT_LOCAT- Position the cursor on the screen. (DT_MISCI.C)

TYPE
Screen Interface

DECLARATION

```
COUNT    DT_LOCAT(col,row)
COUNT    col,row;      /* row and column on screen */
```

DESCRIPTION
This function will position the cursor at the given coordinates, utilizing the escape sequences from the termcap file.

RETURN
NORMAL RETURN
Returns a zero for successful completion.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x2401	ERR_2401	Invalid screen coordinates.

EXAMPLE
DT_LOCAT(10,01); /* position cursor at column 10 on row 1 */

REFERENCE NUMBER - 24

NAME
DT_LOCPT- Position the cursor for stream output. (Locate for print output) (DT_MISCI.C)

TYPE
Screen Interface

DECLARATION

```
COUNT      DT_LOCPT(fp,col,row)
FILE       *fp;          /* output file ptr */
COUNT      col,row;      /* column and row */
```

DESCRIPTION
This function will position the cursor at the given coordinates pertaining to the output stream. It is primarily used when the print screen has been hit.

RETURN
Always returns a zero.

EXAMPLE
DT_LOCPT(fp,10,01); /* position cursor relative to column 10 on row 1 on printed page*/

REFERENCE NUMBER - 75

NAME
DT_MAPDD- given two DODA symbolic names, map one field's contents to another. (DT_WDODA.C)

TYPE
Data Management

DECLARATION
```
COUNT     DT_MAPDD(dsymb,ssymb)
TEXT      *dsymb;     /* destination symbol */
TEXT      *ssymb;     /* source symbol */
```

DESCRIPTION
This function simply determines the addresses associated with the two symbolic names and does a memory copy from the address of the source to the address of the destination.

RETURN
Returns a 1 if function failed.
returns a zero if successful.

EXAMPLE
```
if (DT_MAPDD("myname","yourname"))
        printf("Could not copy yourname to myname");
```

REFERENCE NUMBER - 4E

NAME
DT_MAPIT- execute field maps defined in given MAPIT definition. (DT_MAPIT.C)

TYPE
Data Management

DECLARATION
```
COUNT     DT_MAPIT(mapno)
COUNT     mapno;        /* map definition to execute */
```

DESCRIPTION
Using the MAPIT keyword in a d-tree script, fields that are to be copied to one another (mapped to one another) are defined. This function executes these copies(mapps) for the map definition associated to the given number.

RETURN
NORMAL RETURN
Returns a zero for successful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x961	ERR_961	Could not allocate space for extract.

EXAMPLE
```
if (mapno) DT_MAPIT(mapno);
```

REFERENCE NUMBER - 96

NAME
DT_MENUS- Display MENU and execute menu logic. (DT_MENUS.C)

TYPE
Special Ability

DECLARATION

```
COUNT    DT_MENUS(menusno,display)
COUNT    menusno;    /* menu number to use */
COUNT    display;    /* display menu flag 1=display 0=no display*/
```

DESCRIPTION
This function provides a menu interface for the programmer. MENUS screens and reactions are defined in a d-tree script. This function displays the proper IMAGE, accepts the input and executes the proper logic based on the input.

d-tree script syntax:

```
MENU(master)
USES_IMAGE(menu)
/*      Call Criteria              Type of Call        Call Value */
option=1 CURSOR=name               EXECL               my_program
option=1 CURSOR=name               SYSTEM              my_program
option=1 CURSOR=name               CALL                my_function
option=1 CURSOR=name               RETURN              1
```

This function will react based on the-
Type Of Calls
EXECL - execute execl logic
SYSTEM - execute a system call
CALL - call another function
RETURN - return a value from menu function

RETURN
NORMAL RETURN
Returns value resulting from call to function, program, or menu otherwise, zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x5F1	ERR_5F1	Invalid MENUS number.
0x5F2	ERR_5F2	Invalid Image number.

EXAMPLE

```
switch (DT_MENUS(menuno,1))
        {
        case 1: /* Process menu selection 1. */
        }
```

REFERENCE NUMBER - 5F

NAME
DT_NSERT- Insert a character into a character array. (DT_UTILY.C)

TYPE
Utility

DECLARATION
```
COUNT    DT_NSERT(ch,string,pos)
COUNT    ch;          /* character to insert */
TEXT     *string;     /* character array to insert character into */
COUNT    pos;         /* position in array to insert character */
```

DESCRIPTION
This function inserts a character into a string after the specified position.

EXAMPLE
```
TEXT name[32];
strcpy(name,"abcdefg");
DT_NSERT('x',name,1);
```

yields... "axbcdefg"

REFERENCE NUMBER - 79

NAME
DT_NXREC- Get the next record in a c-tree file. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_NXREC(filno)
COUNT     filno;   /* file number */
```

DESCRIPTION
This function performs the following steps.
a) calls c-tree's NXTREC to get the record. if error on NXTREC return error.
b) determines if the file is variable length or not. If variable length it then unpacks the record into the maintenance buffer otherwise its simply copies the read record into the maintenance buffer.
c) if UNIFRMAT is define the uniformat logic is executed.
d) DT_UNPAD is called to unpack fixed length fields.

RETURN
NORMAL RETURN
Returns a zero for successful get.
ERROR RETURN
Returns the c-tree NXTREC error if failed.

EXAMPLE
```
if (!(DT_NXREC(keyno)))
        printf("Got next record");
```

REFERENCE NUMBER - 93

NAME
DT_OFSET- calculate the offset of a given DODA entry. (DT_ALIGN.C)

TYPE
Doda Managment

DECLARATION

```
UCOUNT   DT_OFSET(ptrdoda,offset)
DATOBJ   *ptrdoda;    /* pointer to doda entry */
UCOUNT   offset;      /* last fields offset */
```

DESCRIPTION
This function is used to determine the offset of a field defined in the DODA. Given the DODA pointer to the field and the offset of the prior field, this function will calulate the offset of the given field and store the offset in the fhrc field in the DODA structure. This function is called by the alignment function DT_ALIGN as it loops thru all the entries in the DODA.

RETURN
Returns the calculated offset.

EXAMPLE

```
offset=DT_OFSET(ptrdoda,offset);
```

REFERENCE NUMBER - 86

NAME
DT_PARSE- Primary Parsing Function. (DT_PARSE.C)

TYPE
Parsing

DECLARATION
```
COUNT     DT_PARSE(scrptnam)
TEXT      *scrptnam;   /* script file name */
```

DESCRIPTION
This function is the primary parsing function whichs reads the d-tree script whose name was passed as the parameter. It scans for valid keywords that were defined in DT_VALID.H. For each valid keyword found it calls the DT_PBUFF function which load a temporary parsing buffer with this keywords definition and then calls the associated keyword parsing routine passing the pointer to this buffer and it's length. Each keyword's own parsing routine interprets this buffer and initializes it's associated typedef.

RETURN
NORMAL RETURN
Returns a zero if successful.

ERROR RETURN
if the parse fails, the error from the keyword parsing routine that failed is returned.

Value	Symbolic Constant	Explanation
1101	ERR-1101	fopen error
1102	ERR-1102	no valid keyword found in script
1103	ERR-1103	unable to allocate temp parsing buffer

EXAMPLE
```
if (err=DT_PARSE(myscript))
        {
        printf("\nError Occured During Parse Error=%x\n",err); exit(1);
        }
```

REFERENCE NUMBER - 11

NAME
DT_PBUFF- Load Temporary Parsing Buffer. (DT_PARSE.C)

TYPE
Parsing

DECLARATION
```
DTTKEYWD  *DT_PBUFF(parsebuf,fp,len,ch)
TEXT      *parsebuf;  /* pointer to parsing buffer */
FILE      *fp;        /* source file pointer */
COUNT     *len;       /* length loaded into buffer */
COUNT     *ch;        /* last char read from text file */
```

DESCRIPTION
The purpose of this function is to load a temporary buffer that is to be used by a specific ability parsing routine. By looping on the DT_TOKEN function the source file is read and each token is checked to see if it is a keyword. If so the loop terminates. Note the option of the DT_TOKEN function. If a buffer address is passed as it's third parameter, that buffer is loaded as it looks for the next keyword. The pointer to this temporary buffer is saved upon entry in order to to calculate the length of the buffer's data upon return. Once a new keyword is found the function will return a pointer to that keyword's structure. Because the address of len was passed, the calling function is able to utilize the calculated length of the temporary buffer. If EOF is hit, a return(0) will notify the calling function that there are no more keywords.

This function is called by the master parsing function DT_PARSE. The primary flow of DT_PARSE is to loop on this function as long as there are keywords. This function loads the temporary parsing buffer, providing it's length and returning a pointer to the next keyword. DT_PARSE then calls the current keyword's associated parsing routine, passing it the loaded buffer address and it's length.

RETURN
0 - Hit EOF no nor keywords in source.
ptr - Keyword stucture pointer to next keyword found in source.

EXAMPLE

```
while (nxtptr)/* loop while keyword are hit */
{
        len=0;          /* reset parsing buffer length */
        nxtptr=DT_PBUFF(parsebuf,fp,&len,&ch); /* load temp parsing buffer */
        if (err=(*ptr-fptr)(ptr,parsebuf,len)) /* call keywords parsing funct */
                return(err);            /* if parse has error then return */
        ptr=nxtptr;                     /* remember we are using function */

        /* pointers here */
}       /* end loop for keyword parsing   */
```

SEE ALSO

DT_PARSE - Primary Parsing Routine.

REFERENCE NUMBER - 14

NAME
DT_PRMPT- File Access Prompt Routine (DT_PRMPT.C)

TYPE
Special Ability

DECLARATION
```
COUNT    DT_PRMPT(prmptno,keyno,scanno,target,tarsigln)
COUNT    prmptno;     /* prompt number to use */
COUNT    *keyno;      /* key number to be used for get function */
COUNT    *scanno;     /* scan number to use */
TEXT     *target;     /* target to be used for get function */
COUNT    *tarsigln;   /* target sig length */
```

DESCRIPTION
This function displays the IMAGE specified in the d-tree script and accepts input from the user. Based on information defined in the script, this function will:
a) Initialize a target (key) value to be used to access a c-tree file, doing all defined
concatenations as well as executing c-tree's TFRMKEY.
b) set tarsigln to the defined significant length for the target.
c) set keyno to the proper index (or data) file number.
d) set scanno to the proper scan to use if there is a non-equal access to the file.

RETURN
NORMAL RETURN
Returns the last keystroke from the keyboard.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x4001	ERR_4001	Prompt number not defined.
0x4002	ERR_4002	Associated IMAGE not found.
0x4003	ERR_4003	Prompt not found in extracted subset.
0x4004	ERR_4004	Target fields not found for target build.
0x4005	ERR_4005	Could not allocate space for extract.

EXAMPLE

```
while ((DT_PRMPT(prompt,&keyno,&scann,target,&tarsigln))!=DTKBESC)
{
        err=0;
        if (keyno && !(DT_EQREC(keyno,target)))
                { err=1; }
        else
                {
                if (!keyno) keyno=datno;
                if (!(DT_SCANN(scann,keyno,target,tarsigln)) ) { err=1; }
                } /* end not hit on eqlrec */
        if (err)
                {
                DTchgmode(mode);
                } /* end got a record */
        DT_DOINT(datno); option=0;
} /* end prompt while */
```

REFERENCE NUMBER - 40

NAME
DT_PSTFX- Convert an expression in infix to postfix form. (DT_PSTFX.C)

TYPE
Utility

DECLARATION
```
COUNT   DT_PSTFX(in,out)
TEXT    *in;    /* pointer to infix expression */
TEXT    *out;   /* pointer to postfix expression */
```

DESCRIPTION
This function will convert an infix expression to a postfix expression. Once an expression is in postfix form, it is easier to evaluate.

RETURN
always returns a zero.

EXAMPLE
given ....

(((A/(B*C)) + (D*E))-(A*C))

DT_PSTFX yeilds...

ABC*/DE* + AC*-

REFERENCE NUMBER - 6A

NAME
DT_PVREC- Get the previous record in a c-tree file. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_PVREC(filno)
COUNT     filno;   /* file number */
```

DESCRIPTION
This function performs the following steps.
a) c-tree's PRVREC to get the record. if error on PRVREC return error.
b) Determines if the file is variable length or not. If variable length it then unpacks the record into the maintenance buffer otherwise it simply copies the read record into the maintenance buffer.
c) if UNIFRMAT is defined the uniformat logic is executed.
d) DT_UNPAD is called to unpack fixed length fields.

RETURN
NORMAL RETURN
Returns a zero for sucessful get.
ERROR RETURN
Returns the c-tree PRVREC error if failed.

EXAMPLE
```
if (!(DT_PVREC(keyno)))
        printf("Got previous record");
```

REFERENCE NUMBER - 94

NAME
DT_RCDLN- given the last potential offset for a file, return the record length. (DT_ALIGN.C)

TYPE
DODA Management

DECLARATION
```
UCOUNT    DT_RCDLN(offset)
UCOUNT    offset;        /* next offset */
```

DESCRIPTION
In the alignment functions, we loop thru the DODA entries determining the offsets of the fields. When the last field is calculated, offset is set to the next byte after the last field's length. By passing this offset to the DT_RCDLN function, the length of the record structure is returned.

RETURN
Returns the offset where the next structure would be aligned or in otherwords the structure length of the previous structure.

EXAMPLE
```
UCOUNT DT_ALDOD(dodaptr,nooflds)
DATOBJ *dodaptr;          /* pointer to first field in record */
COUNT nooflds;            /* number of fields in record */
{
        COUNT DT_STALN();              /* set alignment array */
        UCOUNT DT_OFSET();    /* calc field offsets */
        UCOUNT DT_RCDLN();    /* calc record length */
        COUNT offset=0;
        COUNT noindoda=0;

        DT_STALN();

        while (dodaptr-fwhat!=-1)
                {
                offset=DT_OFSET(dodaptr,offset);
                ++noindoda;       /* count number of entries in doda */
                ++dodaptr;/* next doda */
                if (noindoda==nooflds) break;
                }

        return(DT_RCDLN(offset)); /* return record length */
} /* end function */
```

REFERENCE NUMBER - 87

NAME
DT_RDBUG- Relate Structure Debug. (DT_DEBUG.C)

TYPE
Internal Structure Relationships

DECLARATION
```
COUNT     DT_RDBUG(adr,num)
DTTRELAT  *adr;  /* relate structure base address */
COUNT     num;   /* number of relate occurances */
```

DESCRIPTION
This function is used to display the given relate structure on the screen. The address of the RELAT strucure type as well as the number of elements in the RELAT array are given to this function. It is very useful in debugging programs which use relationships defined by the relate function.

RETURN
always returns a zero.

EXAMPLE
DT_RDBUG(DTGRELAT,DTNRELAT);

REFERENCE NUMBER - 44

NAME
DT_REFMT- Reformat a file. (DT_REFMT.C)

TYPE
Data Managment

DECLARATION
```
COUNT    DT_REFMT(sdoda,sflds,sfilnam,ddoda,dflds,dfilnam)
DATOBJ   *sdoda;      /* source doda */
COUNT    sflds;       /* number of source fields */
TEXT     *sfilnam;    /* source file name */
DATOBJ   *ddoda;      /* destination doda */
COUNT    dflds;       /* number of destination fields */
COUNT    *dfilnam;    /* destination file name */
```

DESCRIPTION
This function will reformat the definition of a c-tree file based on the definition provided in the two DODA's that are passed. This function will reformat file's in place. At this time fixed length files are supported. Variable length file support is being tested.

RETURN
NORMAL RETURN
Returns a zero for sucessful completion.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x6C1	ERR_6C1	Space Allocation error.
0x6C2	ERR_6C2	Could not Open Source file.
0x6C3	ERR_6C3	Doda Length Does not match Source File.
0x6C4	ERR_6C4	Source file not a data file.
0x6C5	ERR_6C5	Source file Corrupt at open.
0x6C6	ERR_6C6	Could not read File header info.
0x6C7	ERR_6C7	Could not read Variable length record six byte header.
0x6C8	ERR_6C8	Source record READ error.
0x6C9	ERR_6C9	Write of Null Header failed after format.

EXAMPLE
```
if ((c=DT_REFMT(oldfile,oldflds,oldnam,newfile,newflds,newnam)))
        return(c);
```

REFERENCE NUMBER - 6C

NAME
DT_RSORT- Sort Relate Structure. (DT_RELAT.C)

TYPE
Internal Structure Relationship

DECLARATION

```
COUNT    DT_RSORT( base, nel, width, mode)
TEXT     *base;
COUNT    nel, width, mode;
```

DESCRIPTION
This function is used to sort the relate typdef structure.

Valid Modes- (defined in DT_TYPDF.H)

```
#define DTQSLTYP 1    /* Sort by left side type. */
#define DTQSLTAC 2    /* Sort by left side type and count. */
#define DTQSLTAA 3    /* Sort by left side type and alt sort. */
#define DTQSLSRT 4    /* Sort by left side alt sort field. */
#define DTQSRTYP 5    /* Sort by right side type. */
#define DTQSRTAC 6    /* Sort by right side type and count. */
#define DTQSRTAA 7    /* Sort by right side type and alt sort. */
#define DTQSRSRT 8    /* Sort by right side alt sort field. */
#define DTQSBAAC 9    /* Sort by left side alt and right side cnt */
#define DTQSBCAC 10   /* Sort by left side cnt and right side cnt */
#define DTQSKYBD 11   /* KEYBD sort-by terminal,key, no of gets */
#define DTQSHOOK 12   /* HOOKS sort-by spot (hook location) */
```

RETURN
Always returns a zero.

EXAMPLE

```
/* Example taken from DTPKEYBD.C - Keyboard Parse routines. */
DT_RSORT(DTGPOINT[DTGCURGP][DTKKEYBD],
         DTGNNUMBR[DTGCURGP][DTKKEYBD],
         sizeof(DTTKEYBD),
         DTQSKYBD);
```

REFERENCE NUMBER - 43

NAME
DT_RTREE- r-tree front end prompt. (DT_RTREE.C)

TYPE
Special Ability

DECLARATION
```
COUNT     DT_RTREE(rtreeno)
COUNT     rtreeno;        /* rtree number to use */
```

DESCRIPTION
This function provides a front end to a r-tree report. The function first displays and accepts input from the IMAGE defined in the d-tree script. Based on the definition in the d-tree script it will then make the proper substitutions into the script and call the defined r-tree program.

RETURN
NORMAL RETURN
normal return is zero, or DTKBESC if the <ESC> key was pressed

ERROR RETURN

Value	Symbolic Constant	Explanation
0x6E1	ERR_6E1	rtree number not defined.
0x6E2	ERR_6E2	associated IMAGE not found.
0x6E3	ERR_6E3	rtree not found in extracted subset.
0x6E5	ERR_6E5	could not allocate space for extract.
0x6E6	ERR_6E6	could not open script work file.
0x6E7	ERR_6E7	could not open base script file.

EXAMPLE
```
/* This will execute the balance sheet report. */

DT_RTREE(DT_INAME("balsheet"));
```

REFERENCE NUMBER - 6E

NAME
DT_RWREC- Add a record to the c-tree data base. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
COUNT   DT_RWREC(datno)
COUNT   datno;/* data file number to add record to. */

DESCRIPTION
Rewrite a record back to a c-tree file.
1) Pad the fields in the record buffer with the PADDING character from c-tree.
2) Execute UNIFORMAT logic if defined by c-tree.
3) Determine the type of file record being written: either fixed or variable length.
4) If variable length file it will pack the record.
5) Enable record locking. LKISAM(ENABLE).
6) Execute c-tree's rewite record call. Either RWTREC or RWTVREC.
7) Free the record lock. LKISAM(FREE).
8) Commercial version will edit for muli-user interface with three buffer approach.

RETURN
NORMAL RETURN
Returns a zero for successful rewrite.
ERROR RETURN
Returns the c-tree isam_err value if rewrite failed.

EXAMPLE
```
if (err=DT_RWREC(datno))
        printf(target,"Unable to rewite record c-tree error = %d",err);
```

REFERENCE NUMBER - 4F

NAME
DT_SCANN- scan records in a c-tree file. (DT_SCANN.C)

TYPE
Special Ability

DECLARATION
```
COUNT     DT_SCANN(scanno,filno,target,tarsigln)
COUNT     scanno;       /* scan number */
COUNT     filno;        /* file number */
TEXT      *target;      /* target for access */
COUNT     tarsigln;     /* sig length of target */
```

DESCRIPTION
This function provides an interface to a c-tree file for scanning the data records. Records are displayed and can be rolled up and down thru the data file. Optionally maintenance may be performed on displayed records. This function provide a facility similar to the "browse" mode of some data base products.

RETURN
NORMAL RETURN
returns a zero if record was selected.
Returns a 1 if not record was selected.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x4101	ERR_4101	Scann number not a defined by SCANN
0x4102	ERR_4102	IMAGE_OUT display failed.
0x4103	ERR_4103	EQLREC failed on prev displayed rcd.

EXAMPLE

```
while ((DT_PRMPT(prompt,&keyno,&scann,target,&tarsigln))! = DTKBESC)
        {
        err=0;
        if (keyno && !(DT_EQREC(keyno,target))) { err=1; } /* end if hit on eqlrec */
        else
                {
                if (!keyno) keyno=datno;
                if (!(DT_SCANN(scann,keyno,target,tarsigln)) ) { err=1; }
                /* err=1 means got record on scann */
                } /* end not hit on eqlrec */
        if (err)
                {
                DTchgmode(mode);
                } /* end got a record */
        DT_DOINT(datno); option=0;
        } /* end prompt while */
```

REFERENCE NUMBER - 41

NAME
DT_SCGET- function used by DT_SCANN to get a selected c-tree record. (DT_SCANN.C)

TYPE
Special Ability

DECLARATION
```
COUNT    DT_SCGET(keyno,datno,recptr,prev)
COUNT    keyno;     /* key number to get from */
COUNT    datno;     /* data file number */
TEXT     *recptr;   /* record buffer to read into */
COUNT    prev;      /* records prev is the record to get */
```

DESCRIPTION
When a record is selected in the scan logic, this function will locate it in the file and read the selected record.

RETURN
NORMAL RETURN
returns a zero if record was found.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x4901	ERR_4901	EQLREC Failed.

EXAMPLE
```
if (DT_SCGET(keyno,datno,recptr,(nosel-1)))
        return(ERR_4103);
```

REFERENCE NUMBER - 49

NAME
DT_SCSEQ- Output a special control sequence. (DT_MISCI.C)

TYPE
Screen I/O

DECLARATION
```
COUNT     DT_SCSEQ(fp,scseq)
FILE      *fp;          /* file for output */
COUNT     scseq;        /* special sequence number */
```

DESCRIPTION
This function outputs to the given file pointer (stdout) the special character sequence define by the given number. Special character sequences are initialized by DTPKEYBD from the termcap file for functions such as clear screen or draw a frame.

RETURN
NORMAL RETURN
returns zero for sucessful completion.
ERROR RETURN
Returns a 1 in scseq is not a valid special sequence number.

EXAMPLE
Here is the clear screen function.

```
COUNT DT_CLEAR()        /* clear screen */
{
        FAST COUNT c;
        FAST DTTKEYBD *kptr;

        kptr=(DTGKEYBD-DTSCCLS+DTSCFRS);
        putchar(kptr-retcode);
        for (c=0; c<kptr->noofchar; ++c)
        {
                putchar(kptr-addchar[c]);
        }
        return(0);
}
```

REFERENCE NUMBER - 76

NAME
DT_SETTY- initialize tty mode (UNIX) (DT_MISCI.C)

TYPE
Keyboard Input

DECLARATION
```
COUNT       DT_SETTY(mode)
FAST COUNT        mode;/* set mode */
```

DESCRIPTION
This function is called at the beginning of a program to set the I/O modes of the tty line on a UNIX system. It can be used for other startup and clean up functions. In the d-tree model programs, DT_SETTY(1) is called at the beginning of each program and DT_SETTY(0) is called at the end of each program.

RETURN
Always returns a zero.

EXAMPLE
```
DT_SETTY(1);
```

REFERENCE NUMBER - 52

NAME
DT_SFCAD- Subfile Child Add Routine. (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
```
COUNT    DT_SFCAD(parent,ocur,child)
COUNT    parent;       /* parent subfile */
COUNT    ocur;  /* parent occurrence */
COUNT    child;  /* parent subfile */
```

DESCRIPTION
This function writes all the records in a child subfile. Looping thru each record in the parent subfile, this function then accesses the subordinate records in the child subfile and executes the subfile record add function DT_SFHAD.

RETURN
NORMAL RETURN
return the number of records added.
ERROR RETURN
returns zero if subfile not found or allocated.
(value of uerr_cod)

Value	Symbolic Constant	Explanation
0x1E01	ERR_1E01	Invalid Parent subfile.
0x1E02	ERR_1E02	Parent not yet allocated.
0x1E03	ERR_1E03	Invalid child subfile.
0x1E04	ERR_1E04	Child not yet allocated.

EXAMPLE
DT_SFCAD(sfl2,ocur,sfl3);

REFERENCE NUMBER - 1E

NAME
DT_SFCLD- Load Child Subfile. (DT_SUBFL.C)

TYPE
Subfile

DECLARATION

```
COUNT   DT_SFCLD(parent,ocur,child,mode)
COUNT   parent;        /* parent subfile */
COUNT   ocur;          /* occurrence of parent */
COUNT   child;         /* parent subfile */
COUNT   mode;          /* subfile load mode */
```

DESCRIPTION
This function loads records into a child subfile. By looping thru the parent subfile, the target value for the child read is obtained. The child subfile is then loaded with proper records.
Subfile load modes:
DTSFLLIN - Subfile load initialize block only.
DTSFLLOD - Subfile load data from disk.

RETURN
NORMAL RETURN
Returns a zero if successful.
ERROR RETURN (value of uerr_cod)

Value	Symbolic Constant	Explanation
0x1D01	ERR_1D01	DT_SFCLD-Invalid Parent subfile
0x1D02	ERR_1D02	DT_SFCLD-Invalid Child subfile
0x1D03	ERR_1D03	DT_SFCLD-Parent not yet allocated

via DT_SFHLD:

Value	Symbolic Constant	Explanation
0x5901	ERR_5901	DT_SFHLD-Invalid subfile number
0x5902	ERR_5902	DT_SFHLD-Could not allocate extract space
0x5903	ERR_5903	DT_SFHLD-Could not allocate memory control block
0x5904	ERR_5904	DT_SFHLD-Occurrences number out of range

EXAMPLE
DT_SFCLD(sfl2,ocur,sfl3,DTSFLLIN);

REFERENCE NUMBER - 1D

NAME
DT_SFHAD- Subfile High Level Add (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
```
COUNT    DT_SFHAD(sflno,ocur)
COUNT    sflno;  /* subfile number */
COUNT    ocur;  /* subfile occurrence */
```

DESCRIPTION
This function loops thru the records in a subfile and calls DT_SFLAD (subfile low level add) for each record. The result is that every record in the subfile will be written to disk.

RETURN
NORMAL RETURN

If uerr_cod = zero then return value is number of records added.

ERROR RETURN (value of uerr_cod)

Value	Symbolic Constant	Explanation
0x6101	ERR_6101	Invalid subfile number.
0x6102	ERR_6102	Subfile not allocated.

via DT_SFLAD:

0x5801	ERR_5801	Could not allocate space for extract.

EXAMPLE
```
DT_SFHAD(sfl1,ocur);
```

REFERENCE NUMBER - 61

NAME
DT_SFHDL- Subfile High Level Delete. (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
COUNT DT_SFHDL(sflno)
COUNT sflno; /* subfile number */

DESCRIPTION
This function will delete all records that were previously loaded into it from the c-tree file. This function is usually called when an array of records was loaded into a subfile, the subfile was maintained, and it is time to update the disk with the changed records in the subfile. The current d-tree approach is to delete all records that were loaded and to rewrite the changes to disk as a series of adds. This function will do the delete.

RETURN
NORMAL RETURN
If uerr_cod = zero, then return value is number of records deleted.
ERROR RETURN (value of uerr_cod)

Value	Symbolic Constant	Explanation
0x6001	ERR_6001	Invalid subfile number.
0x6002	ERR_6002	Subfile not allocated.
0x6003	ERR_6003	No of rcds loaded into sfl not same as the no deleted.

EXAMPLE
DT_SFHDL(sfl2);

REFERENCE NUMBER - 60

NAME
DT_SFHLD- Subfile High Level Load (DT-SUBFL.C)

TYPE
Subfile

DECLARATION

```
COUNT      DT_SFHLD(sflno,ocur,mode)
COUNT      sflno;        /* subfile number */
COUNT      ocur;         /* occurrence */
COUNT      mode;         /* load mode */
```

DESCRIPTION
This function loads records from a c-tree file into a subfile based on the definition provided from the d-tree script. This function allocates the memory space for the subfile and then calls the subfile low level function (DT_SFLLD) to load the records.
Subfile load modes:
DTSFLLIN - subfile load initilize block only.
DTSFLLOD - Subfile load data from disk.

RETURN
NORMAL RETURN
Returns c-tree's uerr_cod if error.
ERROR RETURN (value of uerr_cod)

Value	Symbolic Constant	Explanation
0x5901	ERR_5901	Invalid subfile number.
0x5902	ERR_5902	Could not allocate extract space.
0x5903	ERR_5903	Could not allocate memory control block.
0x5904	ERR_5904	Occur Number out of range.

EXAMPLE
DT_SFHLD(sfl1,0,DTSFLLOD);

REFERENCE NUMBER - 59

NAME
DT_SFLAD- Subfile Low Level Add (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
```
COUNT      DT_SFLAD(ptr,datno,norcds,sfloccur)
DTTSUBSB   *ptr;   /* pointer to memory buffer */
COUNT      datno;/* data file to add record to */
COUNT      norcds;       /* number of records to add */
COUNT      sfloccur;     /* subfile number */
```

DESCRIPTION
This function adds records from a subfile to a c-tree file. As each record in the subfile memory area is processed, the subfile must have (SFL_MUSTHAVE) logic is executed to see if the record should be added. If it passes this test, then the SFL_MAP logic is executed to map data into the record before it is added. DT_ADREC is then called to add the record.

RETURN
NORMAL RETURN
If uerr_cod = zero, then return value is number of records added.
If zero is returned then check uerr_cod for error.
ERROR RETURN (value of uerr_cod)

Value	Symbolic Constant	Explanation
0x5801	ERR_5801	Could not allocate space for extract.

EXAMPLE

Here is the subfile high level function that calls this function.

```
COUNT DT_SFHAD(sflno,ocur)
COUNT sflno; /* subfile number */
COUNT ocur; /* subfile occurrence */
{
        COUNT DT_SFLAD();
        FAST DTTSUBFL *sptr; /* subfile work pointer */
        FAST COUNT c; /* work counter */
        uerr_cod=0;
        sptr=DTGSUBFL;
        for (c=0; c<DTNSUBFL; ++c)
                {
                if (sptr-num==sflno)
                        { c = -1; break; }
                ++sptr;
                }
        if (c!=-1)
                {

                uerr_cod=ERR_6101;
                return(0);

                }
        sptr-sptr=sptr-ctlptr+ocur;
        if (!sptr-sptr || !sptr-sptr-sptr)
                { uerr_cod=ERR_6102; return(0); }
        return(DT_SFLAD(sptr-sptr-sptr,revmap[sptr-keyno],
                sptr-maxrcds,
                ((COUNT) (sptr-DTGSUBFL))
                ));
}
```

REFERENCE NUMBER - 58

NAME
DT_SFLDL- Subfile Low Level Delete (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
```
COUNT     DT_SFLDL(keyno,target,tarsigln)
COUNT     keyno;        /* key number for load */
TEXT      *target;      /* target for c-tree set funct */
COUNT     tarsigln;     /* target significant length */
```

DESCRIPTION
This function deletes a group of related records from a c-tree file. With the passed parameters, this function loops thru the set of related records using c-tree's FRSSET & NXTSET logic to delete the records.

RETURN
If zero is returned, check c-tree's uerr_cod for error.
If uerr_cod is zero the the number of records deleted is returned.

EXAMPLE
```
if ((nofound=DT_SFLDL(sptr-keyno,sptr-sptr-target,sptr-sptr-tarsigln))
            !=sptr-sptr-noofrcds)
        {
        printf("Loaded %d rcds but Deleted %d rcds\n",
            sptr-sptr-noofrcds,nofound); getchar();
        uerr_cod=ERR_6003;
        }
```

REFERENCE NUMBER - 57

NAME
DT_SFLLD- Low Level Subfile Load (DT_SUBFL.C)

TYPE
Subfile

DECLARATION

```
TEXT
*DT_SFLLD(keyno,taret,tarigln,memopt,oldptr,sflrcdln,maxrcds,nofound,sfldatno)
COUNT     keyno;       /* key number for load */
TEXT      *target;     /* target for c-tree set funct */
COUNT     tarsigln;    /* target sig length */
COUNT     memopt;      /* memory option */
TEXT      *oldptr;     /* subfile memory blk ptr */
COUNT     sflrcdln;    /* sfl record length */
COUNT     maxrcds;     /* number of sfl records */
COUNT     *nofound;    /* number of record loaded */
COUNT     sfldatno;    /* sfl dat file number */
```

DESCRIPTION
This function uses c-tree's set functions to access the file and load the memory area with the group of related records.
Subfile memory allocation modes
DTSFLANW - allocate new
(clear old subfile memory block and allocate new block)
DTSFLAMR - allocate more
(leave old memory block active and allocate new block)
DTSFLANO - allocate none
(if there is a memory block clear and reuse it else allocate block)
DTSFLAIN - allocate init
(if there is a memory block clear and reuse it else allocate block then return with ptr - do not load data from disk)

RETURN
NORMAL RETURN
Returns a zero if error occurred and error number is in uerr_cod.
Return the pointer to the memory block upon sucessful completion.
ERROR RETURN (Value of uerr_cod)

Value	Symbolic Constant	Explanation
0x5601	ERR_5601	Could not allocate space for sfl.
0x5602	ERR_5602	Memory Block Option is Invalid.

EXAMPLE
sptr=DT_SFLLD(0,NULL,0,DTSFLAIN,sptr,sflrcdln,maxrcds,&nofound,0);

REFERENCE NUMBER - 56

NAME
DT_SFLOT- Subfile Out (Display a subfile) (DT_SUBFL.C)

TYPE
Subfile

DECLARATION

```
COUNT     DT_SFLOT(sflno,sflnxt,sflrcd,logging,title)
COUNT     sflno;       /* subfile number */
COUNT     sflnxt;      /* subfile child number */
COUNT     sflrcd;      /* subfile record number to start with */
COUNT     logging;     /* image logging special flag */
COUNT     title;       /* display title */
```

DESCRIPTION
This function displays a subfile on the screen. If a child subfile is specified (sflnxt), it is also displayed. If the title flag =1, then the image defined in the d-tree script as a subfile title will be displayed. If the title flag >1, then it is assumed to be an image number and that image is displayed.
Image Logging modes:
DTIMGREG - special-nothing special for image out.
DTIMGLOG - special-displaying from log-do not log.
DTIMGSFL - special-image being displayed is subfile.
DTIMGNOL - special-No Log-do not log image.

RETURN
NORMAL RETURN
Returns the number of records displayed.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x6201	ERR_6201	Invalid subfile number.
0x6202	ERR_6202	Subfile not allocated.
0x6203	ERR_6203	Must pass record number.
0x6204	ERR_6204	Record number greater than max records.
0x6205	ERR_6205	Invalid image number for subfile.
0x6206	ERR_6206	Child subfile number invalid.
0x6207	ERR_6207	Parent rcdno is child occurances.

EXAMPLE
DT_SFLOT(livesfl,livesub,1,DTIMGSFL,1);

REFERENCE NUMBER - 62

NAME
DT_SPTRS- Reset pointers in RELAT structure. (DT_SPTRS.C)

TYPE
Internal Structure Relationships

DECLARATION
COUNT DT_SPTRS()

DESCRIPTION
This function loops thru the RELAT strcutre and resets all the pointers to the associated structures.

RETURN
Always returns a zero.

EXAMPLE

```
CODE = DT_SPTRS();
```

REFERENCE NUMBER - 22

NAME
DT_STALN- Set Alignment Array. (DT_ALIGN.C)

TYPE
Doda Management

DECLARATION
COUNT DT_STALN() /* set alignment array */

DESCRIPTION
This function initializes the alignment array that is used to determine field type alignment based on the hardware.

RETURN
NORMAL RETURN
Returns a zero if successful.
ERROR RETURN

Value	Symbolic Constant	Explanation
0x8801	ERR_8801	COUNT, UCOUNT or POINTER are not correctly sized. Call Faircom.
0x8802	ERR_8802	This machine addresses 32 bit words (not bytes). Call FairCom.
0x8803	ERR_8803	This machine addresses words.(not bytes). Call FairCom.

EXAMPLE

Here is the align doda function.

```
UCOUNT DT_ALDOD(dodaptr,noofflds)
DATOBJ *dodaptr; /* pointer to first field in record */
COUNT noofflds;    /* number of fields in record */
{
        COUNT  DT_STALN();                /* set alignment array */
        UCOUNT DT_OFSET();     /* calc field offsets */
        UCOUNT DT_RCDLN();     /* calc record length */
        COUNT offset=0;
        COUNT noindoda=0;
        DT_STALN();
        while (dodaptr-fwhat!=-1)
                {
                offset=DT_OFSET(dodaptr,offset);
                ++noindoda;        /* count number of entries in doda */
                ++dodaptr;/* next doda */
                if (noindoda==noofflds) break;
                }
        if (noofflds)
                return(DT_RCDLN(offset)); /* return record length */
        else
                return(noindoda+1); /* return number of fields in doda */
        /* add one for terminator */
} /* end function */
```

REFERENCE NUMBER - 88

NAME
DT_SUBFL- Maintain a Subfile (DT_SUBFL.C)

TYPE
Subfile

DECLARATION
```
COUNT     DT_SUBFL(sflno,sflnxt,logging)
COUNT     sflno;        /* subfile number */
COUNT     sflnxt;       /* subfile number */
COUNT     logging;      /* immage logging special flag */
```

DESCRIPTION
This function is used to maintain a subfile. Cursor control for moving from field to field (as well as roll up and roll down) is handled by this function.

RETURN
NORMAL RETURN

Returns the last keystroke from the keyboard.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x6401	ERR_6401	Invalid Subfile Number.
0x6402	ERR_6402	Subfile Not Allocated.
0x6403	ERR_6403	Invalid Image define for Subfile.
0x6404	ERR_6404	Could not allocate temp save space.

EXAMPLE
```
if ( (err=DT_SUBFL(livesfl,livesub,DTIMGSFL) ) = = DTKBESC)
        printf("escape hit from subfile");
```

REFERENCE NUMBER - 64

NAME
DT_SUBIT- Subtract one field from another based on DODA symbol names. (doubles only) (DT_WDODA.C)

TYPE
Data Management

DECLARATION
```
COUNT    DT_SUBIT(dsymb,ssymb)
TEXT     *dsymb;     /* destination symbol */
TEXT     *ssymb;     /* source symbol */
```

DESCRIPTION
Given two DODA symbolic names, subtract the value of the second from the first. This function uses DT_TDODA to get the addresses of the symbol names and then does the subtraction. The result is that the value at the destination symbols's address has now been decremented by the value at the source's address. This function assumes that the symbols being passed are of double type. Use this as an example if other types are needed.

RETURN
NORMAL RETURN
Returns a zero if successful.
ERROR RETURN
Returns a 1 if error.

EXAMPLE
```
if (DT_SUBIT("F1400a","F703a"))
        printf("Could not subtract the two values.");
```

SEE ALSO
DT_TDODA

REFERENCE NUMBER - 5A

NAME
DT_TDODA- Validate token as a symbolic name in DODA. (DT_WDODA.C)

TYPE
Table Validation

DECLARATION
```
DATOBJ    *DT_TDODA(dodaptr,token)
DATOBJ    *dodaptr;    /* pointer to DODA */
TEXT      *token;      /* token to validate */
```

DESCRIPTION
This function simply does a "LOOKUP" on the Data Object Definition Array (DODA) to see if the token passed is defined in the DODA as a variable symbolic name.

RETURN
NORMAL RETURN
Returns pointer to the corresponding DODA entry.
ERROR RETURN
Returns zero if item is not found in the DODA.

EXAMPLE

Example- Check parameter passed to pgm as valid symbol in DODA.

```
#include "DT_DEFIN.H"

char NUMBER[11];
char NAME[26];
char CODE[5];

DATOBJ doda[] = {
        {"Cust_Num",  NUMBER, RTSTRING, 11},
        {"Cust_Name", NAME, RTSTRING, 26},
        {"Cust_Code", CODE, RTSTRING, 5},
        {"","",0,0,-1}
};

void main(argc, argv)
int argc;
char *argv[];
{
        DATOBJ *DT_TDODA(),*ptr;
        TEXT token[32];
        strcpy(token,argv[1]);

        if (ptr=DT_TOKKW(token))
                printf("Symbol %s IS in DODA\n",ptr-fsymb);
        else
                printf("Symbol %s IS NOT in DODA\n",token);
} /* end pgm */
```

REFERENCE NUMBER - 17

NAME
DT_TODAY- Access System Date and Time. (DT_UTILY.C)

TYPE
Utility

DECLARATION
```
COUNT      DT_TODAY(date1,mode)  /* get date in string format */
TEXT       *date1;
COUNT      mode;
```

DESCRIPTION
This function places system date/time information into the first parameter based upon the second parameter (mode). Valid modes are:

- 0 - month-day-year (mmddyy)
- 1 - hour-min-sec (hhmmss)
- 2 - year-mon-day-hour-min-sec (yymmddhhmmss)
- 3 - string form
- 4 - month/day/year (mm/dd/yy)

RETURN
Always reaturns zero.

EXAMPLE
```
TEXT  DATEFLD[8];

DT_TODAY(DATEFLD, 0);
```

REFERENCE NUMBER - 5C

NAME
DT_TOKEN- Get the next token from a Text file. (DT_TOKEN.C)

TYPE
Parsing

DECLARATION

```
COUNT       DT_TOKEN(fp,token,load,buffer,strbuf,ch)
FILE        *fp;            /* file pointer for source */
TEXT        *token;         /* buffer to hold token */
COUNT       load;           /* load temp parse buffer flag */
FAST TEXT   **buffer;       /* ptr to ptr to work buffer */
TEXT        *strbuf;        /* beginning of parseing buffer */
COUNT       *ch;   /* last char read from text file */
```

DESCRIPTION
This function will read the text file pointed to by *fp and place the next token into the variable passed as the second parmeter. By token we mean the next set of characters that are separated on both sides by white space character (each word or symbol). A white space character is one of the following: space, new line ('\n'), carrige return ('\r'), and horizontal tab ('\t'). Comments (/* ... */) are ignored. If an address to a buffer is provided as the last parameter, all characters including white spaces, but not including comments, are placed in the buffer. This function is used by the primary parsing routine (DT_PARSE) and the load parsing buffer routine(DT_PBUFF).

RETURN
This function returns the length of the token found. A value of zero means that end of file (EOF) was reached with no token found.

EXAMPLE

```
/* Example- Print all tokens in sample file FILE.TXT */
#include "stdio.h"
main()
{
        FILE *fp, *fopen();
        COUNT DT_TOKEN();

        if ((fp = fopen("FILE.TXT","r")) == NULL)
                exit(1)

        while (DT_TOKEN(fp,token,0))
        {
                printf("Token=%s\n",token);
        } /* end looping thru tokens */

} /* end pgm */
```

REFERENCE NUMBER - 12

NAME
DT_TOKKW- Validate token as Keyword. (DT_PARSE.C)

TYPE
Parsing

DECLARATION
```
DTTKEYWD  *DT_TOKKW(token)
TEXT      *token;        /* token to check */
```

DESCRIPTION
This function is used to validate keywords. Simply pass the variable to be checked to this function and it will return(0) if it is an invalid keyword or a pointer of type DTTKEYWD that point to the structure occurance in which this keyword was found. The logic scans the valid keywords defined in DT_VALID.H. If the token contains a (x) at the end (as in IMAGE(1)) then (X) is dropped before token is compared.

EXAMPLE
```
/* Example- Check parameter passed to pgm as valid keyword */

#include "DT_DEFIN.H"
#include "DT_VALID.H"

void main(argc, argv)
int argc;
char *argv[];
{
        DTTKEYWD *ptr,DT_TOKKW();
        TEXT token[32];

        strcpy(token,argv[1]);

        if (ptr=DT_TOKKW(token))
                printf("Variable %s IS a valid keyword\n",ptr-keyword);
        else
                printf("Variable %s IS NOT a valid keyword\n",token);

} /* end pgm */
```

REFERENCE NUMBER - 13

NAME
DT_TOKNX- Get the next token from memory buffer. (DT_TOKEN.C)

TYPE
Parsing

DECLARATION

```
COUNT    DT_TOKNX(token,strbuf,endbuf,space,tab,cr,nl,termnl)
TEXT     *token;      /* Field pointer to return token in*/
TEXT     **strbuf;    /* starting pointer into buffer */
TEXT     **endbuf;    /* ending pointer into buffer */
COUNT    *space,*tab,*cr,*nl,*term;      /* ptrs to counters*/
```

DESCRIPTION
This function is used to get the next token from a memory buffer, starting at the strbuf position and terminating it's scan at the endbuf position. As it scans it will count the number of separate white space characters (spaces ,tabs ,carriage returns, and new lines) encountered. A COUNT variable for each one of these must be defined in the calling function and the address of each variable passed. This function will then update these variables with the number of each type encountered until a token was found. Note that spaces and tabs are the count since the last new line was encountered.

The results of this call are the following-

- variable token contains the next token or is NULL.

- the counter variables tell you how many of the associated white spaces were hit before the token was found.(spaces & tabs are re-set to zero when newline is hit).

- the function returns the length of the token or zero for no token found.

- the strbuf pointer has been positioned to the first white space after the token or is equal to endbuf.

RETURN
This function returns the length of the token found. A value of zero means that end buffer pointer was reached with no token found.

EXAMPLE

```
/* Example- Print all tokens in buffer. */

#include "DT_DEFIN.H"

main()
{
COUNT DT_TOKNX();
TEXT buffer[64];
TEXT *strbuf,*endbuf;
COUNT space,tab,cr,nl;    /* pointers to counters */
COUNT len;

        strcpy("This is the Buffer we will look at for tokens",buffer);

        strbuf = buffer;
        endbuf = strbuf + strlen(buffer);

        while (len = DT_TOKNX(token,&strbuf,&endbuf,&space,&tab,&cr,&nl))
        {
                printf("Token = '%s'\n",token);
                printf("Token is %d characters long\n");
                printf("Token is on line %d\n",nl + 1);
                printf("Token is in position %d\n",(tabs*5) + space);
        } /* end looping thru tokens */

} /* end pgm */
```

REFERENCE NUMBER - 15

NAME
DT_TSPLT- Split a token. (DT_TOKEN.C)

TYPE
Parsing

DECLARATION

```
COUNT   DT_TSPLT(token,getocur,addit)
TEXT    *token;      /* token to split */
COUNT   getocur;     /* split flag */
COUNT   addit;       /* add to general table flag */
```

DESCRIPTION
This function will take a token in the form ABCDE(1) and return ABCDE and the 1. This function is used in parsing abilities.

RETURN
This function returns the value found between the parentheses "()". If a symbolic reference if found instead of a numeric value, it is evaluated to determine the proper numeric value.

EXAMPLE

```
ocur = DT_TSPLT(token,1,0);   /* takes CHAR(5) and returns */
                              /* token="CHAR" and ocur=5 */
```

REFERENCE NUMBER - 65

NAME
DT_UNPAD- Unpad a record structure. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT    DT_UNPAD(datno) /* un-pad a record */
COUNT    datno;
```

DESCRIPTION
This function will strip off the padding characters from the end of all string type fields in the specified data file record buffer.
RETURN
Always returns a zero.

EXAMPLE
DT_UNPAD(datno);

REFERENCE NUMBER - 5B

NAME
DT_UNPAK- Unpack one record structure into another. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT    DT_UNPAK(base,end,length,fixed,frmbuf,tobuf)
DATOBJ   *base;      /* starting doda pointer */
DATOBJ   *end;       /* ending doda pointer */
UCOUNT   length;     /* structure length of destination buffer */
UCOUNT   fixed;      /* fixed length portion of record */
COUNT    frmbuf;     /* from buffer number */
COUNT    tobuf;      /* to buffer number */
```

DESCRIPTION
This function is used to unpack a variable length record once it is read into a buffer from disk, into the record maintenance buffer. It is called from DT_VLINN when a variable length record has been read from disk and needs to be unpack-edc.

RETURN
Always returns a zero.

EXAMPLE
```
COUNT DT_VLINN(datno)
COUNT datno;
{
        COUNT DT_UNPAK(),DT_INBUF();

        DT_INBUF(dt_sfp[datno]-fadr+dt_sln[datno],dt_sln[datno]);
        if ( REDVREC(datno,dt_sfp[datno]-fadr+dt_sln[datno],dt_sln[datno]) )
                return(ERR_3201);

        /* unpack from 1 to 0 */
        DT_UNPAK(dt_sfp[datno],dt_efp[datno],dt_sln[datno],(key+datno)-reclen,1,0);

        return(0);
}
```

REFERENCE NUMBER - 33

NAME
DT_VLINN- read and unpack the variable length portion of a c-tree variable length record. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
COUNT     DT_VLINN(datno)
COUNT     datno;
```

DESCRIPTION
When a record is read from a c-tree file, this function is called if the record is a variable length record. The variable length portion of the record is read and the function DT_UNPAK is called to unpack the record for maintenance.

RETURN
ERROR RETURN

Value	Symbolic Constant	Explanation
0x3201	ERR_3201	REDVREC failed. see uerr_cod.

EXAMPLE

Note the d-tree equal record function.

```
COUNT DT_EQREC(keyno,target)
COUNT keyno;       /* key number for get */
TEXT *target;      /* target to use for get */
{
        COUNT DT_UNPAD(); /* un-padd a record */
        VOID cpybuf();
        COUNT EQLREC();
                COUNT DT_VLINN();
        COUNT datno;
        COUNT error;

        datno=revmap[keyno];  /* find data file number for this key */
        if ((error=EQLREC(keyno,target,dt_sfp[datno]-fadr+dt_sln[datno])))
                return(error);

        /* if variable length data */
        if ((key+datno)-clstyp= =VAT_CLOSE)
                DT_VLINN(datno);
        else  /* copy record buffer */
                cpybuf(dt_sfp[datno]-fadr,
                        dt_sfp[datno]-fadr+dt_sln[datno],
                        dt_sln[datno]);

#ifdef UNIFRMAT
        unifrmat(datno);
#endif

        DT_UNPAD(datno); /* un-padd a record */
        return(0);
}
```

REFERENCE NUMBER - 32

NAME
DT_VLOUT- Variable record out function. (DT_CTREE.C)

TYPE
File I/O

DECLARATION
```
UCOUNT    DT_VLOUT(dtsln,dtsfp,dtefp,rcdlen)
UCOUNT    dtsln;          /* unpacked record length */
DATOBJ    *dtsfp;         /* first field doda pointer */
DATOBJ    *dtefp;         /* end or last field doda pointer */
UCOUNT    rcdlen;         /* fixed length portion of record */
```

DESCRIPTION
This function will pack a variable length record in preparation for it being written to disk. Note that the pack record is placed in an allocated memory block which should be freed when you are finished with the packed buffer. See add record logic below.

RETURN
Returns the length of the packed version on the record.

EXAMPLE

Note how function is used here in the DT_ADREC function.

```
COUNT DT_ADREC(datno)
COUNT datno; /* data file number to add record to. */
{
VOID mbfree();
COUNT err; /* error flag */
UCOUNT DT_VLOUT();
COUNT DT_DOPAD();
UCOUNT vlen;

DT_DOPAD(datno); /* padd fields */

#ifdef UNIFRMAT
unifrmat(datno);
#endif

if ((key+datno)-clstyp= =VAT_CLOSE) /* if variable length data */
        {
        vlen=DT_VLOUT(dt_sln[datno],dt_sfp[datno],dt_efp[datno],
                (key+datno)-reclen);

        if (dt_vln= =NULL)
                {
                printf("\n%cDtree addvrec datno=%d dt_vln=%x vlen=%d\n",
                        13,datno,dt_vln,vlen);
                printf("dt_vln has a problem\n");
                getchar();
                } /* end debug */

        if (LKISAM(ENABLE) || ADDVREC(datno,dt_vln,vlen)) err=isam_err;
        else { err=0; mbfree(dt_vln); }
        }
else
        {
        if (LKISAM(ENABLE) || ADDREC(datno,dt_sfp[datno]-fadr))
                err=isam_err;
        else err=0;
        }

LKISAM(FREE);
return(err);
}
```

REFERENCE NUMBER - 31

NAME
DT_XTRCT- Extract a subset of the RELAT structure. (DT_RELAT.C)

TYPE
Internal Structure Relationships

DECLARATION
```
DTTRELAT *DT_XTRCT(adr,elements, /* address of & no in base relate    */
                type,               /* type of relate to extract */
                ltyp,lcnt,lsrt,     /* left side type, count, & sort */
                rtyp,rcnt,rsrt,     /* right side type, count & sort */
                altadr,             /* pointer to secondary relate */
                altelem,            /* num of elements in alt RELAT array */
                ltypchk,lcntchk,lsrtchk,     /* alt set left side */
                rtypchk,rcntchk,rsrtchk,     /* alt set right side */
                srtmod,nofound)     /* sort mode and number found */
DTTRELAT  *adr;         /* address of RELAT to extract from */
COUNT     elements;     /* num of elements in RELAT array */
COUNT     type;         /* relationship type */
COUNT     ltyp;         /* type of left structure */
COUNT     lcnt;         /* left pointer count */
COUNT     lsrt;         /* left structure alt sort */
COUNT     rtyp;         /* type of right structure */
COUNT     rcnt;         /* right pointer count */
COUNT     rsrt;         /* right structure alt sort */
DTTRELAT  *altadr;      /* address of alt RELAT to compare against */
COUNT     altelem;      /* num of elements in alt RELAT array */
COUNT     ltypchk;      /* left type altcheck */
COUNT     lcntchk;      /* left counter alt check */
COUNT     lsrtchk;      /* left sort alt check */
COUNT     rtypchk;      /* right type alt check */
COUNT     rcntchk;      /* right counter alt check */
COUNT     rsrtchk;      /* right sort alt check */
COUNT     srtmod;       /* sort mode */
COUNT     *nofound;     /* no of matches found */
```

DESCRIPTION
The extract function is used to make subsets of the any RELAT block. The address of the RELAT block, along with the number of relates in the block, are the first two parameters. The next seven parameters simply define the direct selection criteria from this base RELAT block (such as relate type, left side type, left side count etc.). If an alt pointer is passed, this means that we are defining additional criteria that must be met before a relate entry is selected and entered into our new subset. This criteria is basically that the selected entry must match an entry in an additional RELAT block based on the six chk flags that are sent as parameters to this function.

```
/* relate extract alt set compare types */
/* left side- ltypchk parm */
/* #define DTXTCLTL 1 left type of selected relate = left type in alt set */
/* #define DTXTCLTR 2 left type of selected relate = right type in alt set */

/* left side- lcntchk parm */
/* #define DTXTCLCL 3 left count of selected relate = left count in altset */
/* #define DTXTCLCR 4 left count of selected relate = right count in altset */

/* left side- lsrtchk parm */
/* #define DTXTCLSL 5 left sort of selected relate = left sort in altset */
/* #define DTXTCLSR 6 left sort of selected relate = right sort in altset */

/* right side- */
/* right side- rtypchk parm */
/* #define DTXTCRTL 7 right type of selected relate = left type in altset */
/* #define DTXTCRTR 8 right type of selected relate = right type in altset */

/* right side- rcntchk parm */
/* #define DTXTCRCL 9 right count of selected relate = left count in altset */
/* #define DTXTCRCR 10 right count of selected relate = right count in altset */

/* right side- rsrtchk parm */
/* #define DTXTCRSL 11 right sort of selected relate = left sort in altset */
/* #define DTXTCRSR 12 right sort of selected relate = right sort in altset */
```

RETURN

NORMAL RETURN

Returns a pointer to the subset.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x4501	ERR_4501	Could not allocate extract space

EXAMPLE

```
if (!(rptr=DT_XTRCT(DTGRELAT,DTNRELAT, /* ptr to RELAT & no of elements */
        DTXTCNUL,                      /* relationship type */
        DTKIMAGE,c,DTXTCNUL,           /* IMAGE type and image count */
        DTKFIELD,DTXTCNUL,DTXTCNUL,    /* FIELD type */
        ((DTTRELAT *)(NULL)),DTXTCNUL, /* alt RELAT adr & no elements */
        DTXTCNUL,DTXTCNUL,DTXTCNUL,    /* left side alt criteria */
        DTXTCNUL,DTXTCNUL,DTXTCNUL,    /* right side alt criteria */
        DTQSRTAC,&nofound)) && uerr_cod) /* sort mode and num extracted */
        {
        printf("extract err\n"); getchar();
        return(ERR_4005);              /* memory allocation error */
        }
```

REFERENCE NUMBER - 45

NAME
DT_ZROIT- Zero out a field give only the DODA symbol name. (DT_WDODA.C)

TYPE
Data Management

DECLARATION

```
COUNT    DT_ZROIT(dsymb)
TEXT     *dsymb;     /* destination symbol */
```

DESCRIPTION
This function will zero out a field given the symbolic name for the field in the DODA. This function is only set to handle doubles at this time, but may be used as an example of accessing the DODA.

RETURN
Returns a 1 if failed.
returns a zero if successful

EXAMPLE

```
DT_ZROIT("mybalance");
```

REFERENCE NUMBER - 5D

NAME
DTPCALCS- Parse the CALCS Ability definition. (DTPCALCS.C)

TYPE
Ability Parsing

DECLARATION

```
COUNT     DTPCALCS(kwptr,parsebuf,len)
DTTKEYWD        *kwptr;      /* ptrs to keyword structure */
FAST TEXT       *parsebuf;   /* pointer to temp parsing buffer */
COUNT           len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the CALCS keyword syntax from the temporary parsing buffer and initializes a structure of DTTCALCS type with the definition.

RETURN
NORMAL RETURN
successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x821	ERR_821	Space allocation error
0x822	ERR_822	Expression too long for postfix conversion - change DTMXWORK in DTPCALCS.C

SEE ALSO
DTTCALCS - typedef definition in Ability typdef section.

REFERENCE NUMBER - 82

NAME
DTPCONST- Parse the CONST Ability definition. (DTPCONST.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPCONST(kwptr,parsebuf,len)
DTTKEYWD          *kwptr;       /* ptrs to keyword structure */
FAST TEXT         *parsebuf;    /* pointer to temp parsing buffer */
COUNT             len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the CONST keyword syntax from the temporary parsing buffer and initializes a structure of DTTCONST type which contains the definition. The CONST structure has had all of its fields defined by the IMAGE parse, except the output attribute. This keyword is supplied by this attribute's definition.

ABILITY SYNTAX -

```
CONST(master)
1  RI
^  ^..... output attribute.
............. constant number.
```

ABILITY TYPEDEF -

```
typedef struct {
        TEXT        *string;       /* pointer to constant text */
        COUNT       len;           /* length displayed on screen */
        COUNT       outatr[DT_MXOAT];       /* output attribute */
        COUNT       col;           /* column number for display */
        COUNT       row;           /* row number for display */
} DTTCONST;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x4601	ERR_4601	Image number for constant not defined.
0x4602	ERR_4602	Token not a valid constant or attribute.
0x4603	ERR_4603	Invalid output attribute.
0x4604	ERR_4604	Attribute does not refer to any constant.
0x4605	ERR_4605	Could not allocate space for extract.

SEE ALSO

DTTCONST - typedef definition in Ability typdef section.

REFERENCE NUMBER - 46

NAME
DTPDFALT- Parse the DFALT Ability definition. (DTPDFALT.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPDFALT(kwptr,parsebuf,len)
DTTKEYWD          *kwptr;        /* ptrs to keyword structure */
FAST TEXT         *parsebuf;     /* pointer to temp parsing buffer */
COUNT             len;           /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the DFALT keyword syntax from the temporary parsing buffer and initializes a structure of DTTDFALT type with the definition. An entry is also made into the relate structure to link this default definition to the field that it pertains to.

```
ABILITY SYNTAX -
DEFAULTS(master)
/*      Symbol Name       Type of defaults     Defaults value */
        fiel_name              TAB             Reno Office
        cou_office             INIT            SYSDATE
        cou_office             TAB             SYSTIME

/* Valid DFALT type symbols */
        TAB             - default when auto dup key is hit
        INIT            - default at initialization time
        DUPTAB          - auto dup when auto dup key is hit
        DUPINIT         - auto dup at initialization time

ABILITY TYPEDEF -

typedef struct {
        COUNT       num;          /* default number */
        COUNT       dftyp;        /* type of default */
        TEXT        *dftxt;       /* pointer to default text */
} DTTDFALT;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x7701	ERR_7701	Could not allocate DFALT structure space.
0x7702	ERR_7702	Could not allocate memory for DFALT text.
0x7703	ERR_7703	Could not allocate DFALT's RELAT space.
0x7704	ERR_7704	Error-Default type must follow field name.

SEE ALSO

DTTDFALT typedef definition in Ability typdef section.

REFERENCE NUMBER - 77

NAME
DTPEDITS- Parse the EDITS Ability definition. (DTPEDITS.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT      DTPEDITS(kwptr,parsebuf,len)
DTTKEYWD         *kwptr;      /* ptrs to keyword structure */
FAST TEXT        *parsebuf;   /* pointer to temp parsing buffer */
COUNT            len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the EDITS keyword syntax from a d-tree script and initializes a structure of DTTEDITS type with the definition.

ABILITY SYNTAX -

```
EDITS(master)
Must Enter County Code  cou_cod MAND_FILL
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT    edttyp;      /* type of edit */
        TEXT     *edttxt;     /* pointer to edit message text */
        TEXT     *addtxt;     /* additional text */
} DTTEDITS;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x6601	ERR_6601	Could not allocate EDITS structure.
0x6602	ERR_6602	Could not allocate memory for EDITS text.
0x6603	ERR_6603	Could not allocate EDITS's RELAT memory.
0x6604	ERR_6604	Edit type must follow edit text.
0x6605	ERR_6605	Edit Type must follow field name.
0x6606	ERR_6606	Must enter message text.
0x6607	ERR_6607	Field not found on associated image.
0x6608	ERR_6608	DUPKEY edit must have key no or name.
0x6609	ERR_6609	DUPKEY key symbol length to short. for example: key symbol = "ky" and the key number it represents is "100". The "ky" is only 2 digits, the "100" is 3 digits. Due to memory allocation this is invalid.

SEE ALSO

DTTEDITS - typedef definition in Ability typdef section.

REFERENCE NUMBER - 66

NAME
DTPFIELD- Parse the FIELD Ability definition. (DTPFIELD.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPFIELD(kwptr,parsebuf,len)
DTTKEYWD        *kwptr;      /* ptrs to keyword structure */
FAST TEXT       *parsebuf;   /* pointer to temp parsing buffer */
COUNT           len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the FIELD keyword syntax from the temporary parsing buffer and initializes a structure of DTTFIELD type with the definition.

ABILITY SYNTAX -
```
FIELD(master)
/* Symbol Name  Input Attribute  Output Attribute  Input Order    Special */
     cou_cod        ALLCAPS            RI               1        user_function
```

ABILITY TYPEDEF -

```
typedef struct {
        DATOBJ     *fdoda;        /* pointer to doda */
        COUNT      fdodano;       /* doda number */
        COUNT      len;           /* length displayed on screen */
        COUNT      inpatr;        /* input attribute */
        COUNT      outatr[DT_MXOAT];        /* output attribute */
        COUNT      col;           /* column number for display */
        COUNT      row;           /* row number for display */
        COUNT      dec;           /* decimal positions */
        DT_FPTR    funcptr;       /* special function pointer */
} DTTFIELD;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x1001	ERR_1001	Symbolic Name Not Found In DODA.
0x1002	ERR_1002	Invalid Input Attribute.
0x1003	ERR_1003	Invalid Output Attribute.
0x1004	ERR_1004	Syntax Error Found.
0x1005	ERR_1005	Pointer not pointing to a valid field.
0x1006	ERR_1006	Could not Allocate Parse space.
0x1007	ERR_1007	No Image with same number found.
0x1008	ERR_1008	No Field to Image relationships.
0x1009	ERR_1009	Relationship pointer Incremented to Far.
0x1010	ERR_1010	Invalid user defined function or Invalid field symbol name.

SEE ALSO

DTTFIELD - typedef definition in Ability typdef section.

REFERENCE NUMBER - 10

NAME
DTPHELPP- Parse the HELPP Ability definition. (DTPHELPP.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT      DTPHELPP(kwptr,parsebuf,len)
DTTKEYWD         *kwptr;       /* ptrs to keyword structure */
FAST TEXT        *parsebuf;    /* pointer to temp parsing buffer */
COUNT            len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the HELPP keyword syntax from the temporary parsing buffer and initializes a structure of DTTHELPP type with the definition.

RETURN
NORMAL RETURN
Returns a zero if no errors where found.

ERROR RETURN

Value	Symbolic Constant	Explanation
0x841	ERR_841	DTPHELPP-Space Allocation Error
0x842	ERR_842	DTPHELPP-Syntax Error. Help text or token must be defined before fields
0x843	ERR_843	DTPHELPP-Syntax Error. USES_SFL not define correctly

SEE ALSO
DTTHELPP - typedef definition in Ability typdef section.

REFERENCE NUMBER - 84

NAME
DTPIFILS- Parse the IFILS Ability definition. (DTPIFILS.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT      DTPIFILS(kwptr,parsebuf,len)
DTTKEYWD        *kwptr;      /* ptrs to keyword structure */
FAST TEXT       *parsebuf;   /* pointer to temp parsing buffer */
COUNT           len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the IFILS keyword syntax from the temporary parsing buffer and initializes a structure of DTTIFILS type with the definition.

ABILITY SYNTAX -

```
IFILS
        FILE_NAME ray.dta

        KEY_NAME rayidx
        KEY_FIELDS cou_cod cou_name

        KEY_NAME billidx
        KEY_FIELDS cou_name DUPS_OK

        FIRST_FIELD cou_cod
        LAST_FIELD cou_type
        FIRST_VLEN cou_name
```

ABILITY TYPEDEF - from c-tree

```
typedef struct iseg {
        COUNT      soffset,      /* segment offset */
                   slength,      /* segment length */
                   segmode;      /* segment mode */
} ISEG;

typedef struct iidx {
        COUNT      ikeylen,      /* key length */
                   ikeytyp,      /* key type */
                   ikeydup,      /* duplicate flag */
                   inulkey,      /* null key flag */
                   iempchr,      /* empty character */
                   inumseg;      /* number of segments */
        ISEG       *seg;         /* segment information */
        TEXT       *ridxnam;     /* r-tree symbolic name */
} IIDX;

typedef struct ifil {
        TEXT       *pfilnam;     /* file name (w/o ext) */
        COUNT      dfilno;       /* data file number */
        UCOUNT     dreclen;      /* data record length */
        UCOUNT     dxtdsiz;      /* data file ext size */
        COUNT      dfilmod;      /* data file mode */
        COUNT      dnumidx;      /* number of indices */
        UCOUNT     ixtdsiz;      /* index file ext size */
        COUNT      ifilmod;      /* index file mode */
        IIDX       *ix;          /* index information */
        TEXT       *rfstfld;     /* r-tree 1st fld name */
        TEXT       *rlstfld;     /* r-tree last fld name */
        COUNT      tfilno;       /* temporary file number*/
} IFIL;
```

RETURN
NORMAL RETURN
successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x801	ERR_801	Unable to allocate IFILS structures.
0x802	ERR_802	Unable to allocate IIDXS structures.
0x803	ERR_803	Unable to allocate ISEGS structures.
0x804	ERR_804	Unable to allocate space for text info.
0x805	ERR_805	Syntax error-Must have IFILS symbol.
0x806	ERR_806	Field defined as key segment not in DODA.
0x807	ERR_807	Field defined first or last field not in DODA.
0x808	ERR_808	Field defined as first variable length field is not in DODA.
0x809	ERR_809	Must define KEY_NAME before defining DUPS_OK.

SEE ALSO
DTTIFILS - typedef definition in Ability typdef section.

REFERENCE NUMBER - 80

NAME
DTPIMAGE- Parse the IMAGE Ability definition. (DTPIMAGE.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT      DTPIMAGE(kwptr,parsebuf,len)
DTTKEYWD        *kwptr;      /* ptrs to keyword structure */
FAST TEXT       *parsebuf;   /* pointer to temp parsing buffer */
COUNT           len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the IMAGE keyword syntax from the temporary parsing buffer and initializes a structure of DTTIMAGE type with the definition.

ABILITY SYNTAX -

```
IMAGE(heading)
        {LSTFLD_ADVANCE}
        {FRSFLD_BACKUP}
        {INPUT_ADVANCE=2}

@DATE
@TIME

                    Number: _________
                    Name: ________________________
                    Code: ____
```

ABILITY TYPDEFS -

```
typedef struct {
        COUNT      num;        /* image number */
        COUNT      cls;        /* clear screen flag */
        COUNT      lstcr;      /* exit on last field carride return */
        COUNT      fstbu;      /* exit on first field backup */
        COUNT      inpno;      /* if this many fields have been entered then exit */
        COUNT      topcol;     /* Top left corner column for display */
        COUNT      toprow;     /* Top left corner row for display */
        COUNT      basecol;    /* Top left corner column from parse */
        COUNT      baserow;    /* Top left corner row from parse */
        COUNT      noofvar;    /* number of variable fields */
        COUNT      noofcon;    /* number of constant fields */
        COUNT      fstrow;     /* first row */
        COUNT      lstrow;     /* last row */
        COUNT      lftcol;     /* left most column */
        COUNT      ritcol;     /* right most column */
        TEXT       *varptr;    /* first variable relate ptr */
        TEXT       *conptr;    /* first constant relate ptr */
} DTTIMAGE;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x1601	ERR_1601	Could not Allocate Parse space.
0x1602	ERR_1602	Invalid Optional Feature.
0x1603	ERR_1603	Could not allocate space for DODA.
0x1604	ERR_1604	Could not allocate space for DODA symbolic names text.
0x1605	ERR_1605	INPUT_ADVANCE option invalid.

SEE ALSO

DTTIMAGE - typedef definition in Ability typdef section.

REFERENCE NUMBER - 16

NAME
DTPKEYBD- Parse the KEYBD Ability definition. (DTPKEYBD.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPKEYBD(kwptr,parsebuf,len)
DTTKEYWD          *kwptr;       /* ptrs to keyword structure */
FAST TEXT         *parsebuf;    /* pointer to temp parsing buffer */
COUNT             len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the KEYBD keyword syntax from the temporary parsing buffer and initializes a structure of DTTKEYBD type with the definition. It is normally called by the DT_KEYBD routine which loads the definition into the parsing buffer.

ABILITY SYNTAX -

```
TERMINAL(vt100)
ESC 27 27 CR 13 BU 8 DC 27 91 67 IC 27 91 68 DW 27 91 66
UP 27 91 65 PD 2 PU 6 LF 27 79 82 RT 27 79 83 HM 27 79 80
EN 27 79 81 AD 9 F1 27 49 F2 27 50 F3 27 51 F4 27 52
F5 27 53 F6 27 54 F7 27 55 F8 27 56 F9 27 57 F10 27 58
CTLA 1 F11 3 LOC 0 27 91 120 59 121 72 CLS 27 91 50 74
EOL 27 91 75 UL 27 91 52 109 RI 27 91 55 109 NA 27 91 48 109
PS 2 HL 27 61 LOTUS 27 91 55 109
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT      terminal;      /* terminal id */
        COUNT      retcode;       /* what to return if input matches*/
        COUNT      frschar;       /* first char of key sequence */
        COUNT      noofchar;      /* no of additional char in sequence*/
        TEXT       addchar[DT_MXSEQ];     /* additional chrcters in seqence */
} DTTKEYBD;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x7101	ERR_7101	Could not allocate space for key seq.
0x7102	ERR_7102	Syntax error in termcap definition.
0x7103	ERR_7103	DT_MXSEQ not large enough in DT_TYPDF.H.

SEE ALSO

DTTKEYBD - typedef definition in Ability typdef section.

REFERENCE NUMBER - 71

NAME
DTPKEYST- Keyboard definition sort function. Sort the KEYBD memory block. (DTPKEYBD.C)

TYPE
Ability Parsing

DECLARATION
COUNT DTPKEYST()

DESCRIPTION
This function sorts the keyboard structure by terminal, first key, no of gets.

Explanation- here we first sort the terminal definition in terminal, first key in special sequence, then no of extra getchar's order. We then loop thru the special keys array and map the return code into the keymap array. If any negative numbers are entered, they are mapped above 127 in the keymap array. If we find a sequence that needs more than one getchar or we find more than one sequence that start with the same character, we store the offset + 1000 of the first special sequence in the keymap. When this key is entered we can tell that we need to go to the special definition because the value in the keymap is > 1000. By subtracting 1000 from any value found over 1000 we get the occurrence number of the first special keystroke with this first character.

RETURN
Always returns a zero.

EXAMPLE
DTPKEYST();

REFERENCE NUMBER - 74

NAME
DTPMAPIT- Parse the MAPIT Ability definition. (DTPMAPIT.C)

TYPE
Ability Parsing

DECLARATION

```
COUNT       DTPMAPIT(kwptr,parsebuf,len)
DTTKEYWD         *kwptr;      /* ptrs to keyword structure */
FAST TEXT        *parsebuf;   /* pointer to temp parsing buffer */
COUNT            len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the MAPIT keyword syntax from the temporary parsing buffer and initializes a structure of DTTMAPIT type with the definition. This ABILITY does not have an associated typdef for it simply makes entries into the relate structure to link the two fields.

ABILITY SYNTAX -

```
MAP(name)
/* source field     desination field     length */
        acctnum     cou_cod              3
        acctnam     cou_name
        acctcmt     cou_office
```

RETURN
NORMAL RETURN
successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x951	ERR_951	Must have an even number of flds defined.
0x952	ERR_952	Could not allocate space for MAP relates.
0x953	ERR_953	Field symbol name not found in DODA.
0x954	ERR_954	Copy length longer than child field.
0x955	ERR_955	Invalid MAP type.
0x961	ERR_961	Could not allocate space for extract.

SEE ALSO
DTTMAPIT - typedef definition in Ability typdef section.

REFERENCE NUMBER - 95

NAME
DTPMENUS- Parse the MENUS Ability definition. (DTPMENUS.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPMENUS(kwptr,parsebuf,len)
DTTKEYWD         *kwptr;      /* ptrs to keyword structure */
FAST TEXT        *parsebuf;   /* pointer to temp parsing buffer */
COUNT            len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the MENUS keyword syntax from the temporary parsing buffer and initializes a structure of DTTMENUS type with the definition.

ABILITY SYNTAX -

```
MENU(master)
USES_IMAGE(menu)
/*  Call Criteria              Type of Call     Call Value */
option = 1 CURSOR = name       EXECL            my_program
option = 1 CURSOR = name       SYSTEM           my_program
option = 1 CURSOR = name       CALL             my_function
option = 1 CURSOR = name       RETURN           1
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT     num;         /* menu number */
        COUNT     imageno;     /* image number */
        COUNT     inputfld;    /* input field no */
        COUNT     cursfld;     /* last field that cursor was on */
        COUNT     comptyp;     /* compare type */
        COUNT     calltyp;     /* type of menu call */
        TEXT      *comptxt;    /* compare input field text */
        TEXT      *calltxt;    /* call text */
} DTTMENUS;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x5E1	ERR_5E1	Space Allocation Error.
0x5E2	ERR_5E2	Menu Call type must follow input criteria.
0x5E3	ERR_5E3	Must enter Compare Criteria.
0x5E4	ERR_5E4	Must enter Call Text.
0x5E5	ERR_5E5	CURSOR=symbol..symbol not in DODA.
0x5E6	ERR_5E6	Field compare symbol not in DODA.
0x5E7	ERR_5E7	Must define USES_IMAGE first.

SEE ALSO

DTTMENUS - typedef definition in Ability typdef section.

REFERENCE NUMBER - 5E

NAME
DTPPRMPT- Parse the PRMPT Ability definition. (DTPPRMPT.C)

TYPE
Ability Parsing

DECLARATION

```
COUNT       DTPPRMPT(kwptr,parsebuf,len)
DTTKEYWD          *kwptr;       /* ptrs to keyword structure */
FAST TEXT         *parsebuf;    /* pointer to temp parsing buffer */
COUNT             len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the PRMPT keyword syntax from the temporary parsing buffer and initializes a structure of DTTPRMPT type with the definition.

ABILITY SYNTAX -

```
PROMPT(master)
        USES_IMAGE(prompt)
/* key symbol name      scann name   fields for target     prefix */
        C_Num              master      cou_cod
        C_Nam              master      cou_name               ray
        NONE               master      option
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT       num;          /* prompt number */
        COUNT       imageno;      /* image number */
        COUNT       scanno;       /* associated scann number */
        TEXT        *string;      /* prefix */
} DTTPRMPT;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x3801	ERR_3801	Could not allocate space for prompt.
0x3802	ERR_3802	Could not allocate space for relations.
0x3803	ERR_3803	Invalid keyword-define USES_IMAGE(?).
0x3804	ERR_3804	Image number/name not defined correctly.
0x3805	ERR_3805	Key symbol name not in ISAM definition.
0x3806	ERR_3806	Field symbol name not found in doda.
0x3807	ERR_3807	FIELD defined not on IMAGE defined.
0x3808	ERR_3808	Could not allocate space for prefix.
0x3809	ERR_3809	Scann Number not defined.
0x3810	ERR_3810	Scann Number must be defined before target fields.

SEE ALSO

DTTPRMPT - typedef definition in Ability typdef section.

REFERENCE NUMBER - 38

NAME
DTPRTREE- Parse the RTREE Ability definition. (DTPRTREE.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT      DTPRTREE(kwptr,parsebuf,len)
DTTKEYWD          *kwptr;      /* ptrs to keyword structure */
FAST TEXT         *parsebuf;   /* pointer to temp parsing buffer */
COUNT             len;         /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the RTREE keyword syntax from the temporary parsing buffer and initializes a structure of DTTRTREE type with the definition.

ABILITY SYNTAX -

```
RTREE(my_report)
        USES_IMAGE(my_report)
        USES_SCRIPT(ex_rtree.rts)
        REPORT_PROGRAM(ex_rtree.rts)
        CALL_TYPE(MEMORY)
        CALL_TYPE(EXECL)
        CALL_TYPE(SYSTEM)


/* r-tree              criteria       Substitute */
/* keyword    fields            String */

SEARCH                 NONE           FILE "ARAY.DTA" ALL
option1 FILE "ARAY.DTA" USING KEY KEY1 [ "{option1}"

SELECT                 NONE           ALL
option5 (balance0.00)

VIRTUAL                NONE           dev INT2 2 1
option6       dev INT2 2 "{option6}"

SORT          NONE          LEAVE_OUT
option7       NO_MOD "{option7}"
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT     num;        /* rtree definition number */
        COUNT     imageno;    /* image number  */
        COUNT     type;       /* interface type */
        COUNT     rtkeywd;    /* rtree keyword reference number */
        TEXT      *script;    /* base r-tree script name */
        TEXT      *string;    /* substitute string */
        TEXT      *program;   * program to run */
} DTTRTREE;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

	Value	Symbolic Constant Explanation
0x6F1	ERR_6F1	Could not allocate space for rtree definition.
0x6F2	ERR_6F2	Could not allocate space for relations.
0x6F3	ERR_6F3	Invalid definition keyword -define USES_IMAGE(?) or -define USES_SCRIPT(?).
0x6F4	ERR_6F4	Must define USES_IMAGE and USES_SCRIPT properly. Check for valid image number or that you have not defined both keywords properly.
0x6F5	ERR_6F5	Invalid r-tree keyword.
0x6F6	ERR_6F6	Field symbol name not found in doda.
0x6F7	ERR_6F7	FIELD defined not on IMAGE defined.
0x6F8	ERR_6F8	Could not allocate space for text.
0x6F9	ERR_6F9	One of the following keyword has a syntax error: USES_IMAGE(??) USES_SCRIPT(??) REPORT_PROGRAM(??) CALL_TYPE(??) */
0x6FA	ERR_6FA	Must define substitution text.
0x6FB	ERR_6FB	Invalid Call Type.

SEE ALSO

DTTRTREE - typedef definition in Ability typdef section.

REFERENCE NUMBER - 6F

NAME
DTPSCANN- Parse the SCANN Ability definition. (DTPSCANN.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPSCANN(kwptr,parsebuf,len)
DTTKEYWD        *kwptr;       /* ptrs to keyword structure */
FAST TEXT       *parsebuf;    /* pointer to temp parsing buffer */
COUNT           len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the SCANN keyword syntax from the temporary parsing buffer and initializes a structure of DTTSCANN type with the definition.

ABILITY SYNTAX -

```
SCAN(master)
        {IMAGE_OUT=heading}
        {IMAGE_ROL=rollpart}
        {IMAGE_INP=heading}
        {USE_SETS=2}
```

ABILITY TYPEDEF -

```
typedef struct {
        COUNT      num;        /* scan number */
        COUNT      imgout;     /* header image number */
        COUNT      imgrol;     /* rolling image number */
        COUNT      imginp;     /* input image number */
        COUNT      useset;     /* use c-tree FRSSET logic */
                               /* values 0 = do not use sets */
                               /*  -1 = use provided target sig len */
                               /* >0 = the sig length to use for sets*/
} DTTSCANN;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x3901	ERR_3901	Could not allocate space for SCANN.
0x3902	ERR_3902	Invalid SCANN option defined.
0x3903	ERR_3903	IMAGE_OUT image no not a defined IMAGE.
0x3904	ERR_3904	Must have valid IMAGE_ROL imageno.
0x3905	ERR_3905	Must have valid IMAGE_INP imageno.

SEE ALSO

DTTSCANN - typedef definition in Ability typdef section.

REFERENCE NUMBER - 39

NAME
DTPSUBFL- Parse the SUBFL Ability definition. (DTPSUBFL.C)

TYPE
Ability Parsing

DECLARATION
```
COUNT       DTPSUBFL(kwptr,parsebuf,len)
DTTKEYWD         *kwptr;       /* ptrs to keyword structure */
FAST TEXT        *parsebuf;    /* pointer to temp parsing buffer */
COUNT            len;          /* length of temp parsing buffer */
```

DESCRIPTION
This function interprets the SUBFL keyword syntax from the temporary parsing buffer and initializes a structure of DTTSUBFL type with the definition.

ABILITY SYNTAX -

```
SUBFILE(master)

SFL_IMAGE(rollpart)
SFL_TITLE(title)
SFL_RECORDS(40)
SFL_LINES(18)

SFL_ATTR
NO_ROLL_CR

SFL_TARGET
/* key symbol name      fields for target     prefix */
        C_Num           targetfield(12)       01

SFL_BOUNDARY
/* doda first field    doda last field */
        dodafield1          dodafield2

SFL_MAP
/* parent field    child field    length */
        cou_cod    cou_name       11
        SFL_SEQ    cou_seq

SFL_MUSTHAVE
cou_cod cou_name
```

ABILITY TYPEDEF -

```
typedef struct {
        TEXT        *sptr;              /* sfl memory block */
        TEXT        target[MAXLEN];     /* target used to load sfl */
        COUNT       tarsigln;           /* target sig length */
        COUNT       noofrcds;           /* number of records in sfl */
        COUNT       currcd;             /* current sfl record */
        COUNT       currow;             /* current sfl row */
} DTTSUBSB;

typedef struct {
        COUNT       num;                /* subfile number */
        COUNT       imageno;            /* image number  */
        COUNT       title;              /* title image number  */
        TEXT        *prefix;            /* target prefix */
        COUNT       maxrcds;            /* max number of records for subfile */
        COUNT       sfllines;           /* total number of display lines for sfl */
        COUNT       startdoda;          /* starting doda occurrence number */
        COUNT       enddoda;            /* ending doda occurrence number */
        COUNT       keyno;              /* keyno associated with this subfile */
        COUNT       noofsfls;           /* number of sfl ptrs in memory ctl block*/
        COUNT       sflatr;             /* subfile attributes */
        DTTSUBSB    *ctlptr;            /* sfl memory control block */
        DTTSUBSB    *sptr;              /* current block */
} DTTSUBFL;
```

RETURN

NORMAL RETURN

successful parse returns zero

ERROR RETURN

Value	Symbolic Constant	Explanation
0x5501	ERR_5501	Could not allocate space for subfile definition.
0x5502	ERR_5502	Could not allocate space for relationships.
0x5503	ERR_5503	Parse is expecting to see a subfile keyword-syntax error.
0x5504	ERR_5504	Parse cannot determine what sfl keyword is being parsed-syntax error.
0x5505	ERR_5505	SFL_IMAGE number/name not defined correctly.
0x5506	ERR_5506	Invalid Number for MAX_RECORDS value.
0x5507	ERR_5507	Invalid Number for SFL_LINES value.
0x5508	ERR_5508	SFL_IMAGE must be define before SFL_TARGET.
0x5509	ERR_5509	SFL_RECORDS must be define before SFL_TARGET.
0x5510	ERR_5510	SFL_LINES must be define before SFL_TARGET.
0x5511	ERR_5511	Key symbol name not in ISAM definition.
0x5512	ERR_5512	Syntax error-looking for Key symbol name.
0x5513	ERR_5513	Field symbol name not found in doda.
0x5514	ERR_5514	Only one SFL_TARGET definition allowed.
0x5515	ERR_5515	SFL_MAP Field symbol name not found in doda.
0x5516	ERR_5516	SFL_MAP length longer than child field.
0x5517	ERR_5517	SFL_TARGET must define target field or prefix.
0x5518	ERR_5518	SFL_MUSTHAVE field not found in DODA.
0x5519	ERR_5519	SFL_BOUNDARY field not found in DODA.
0x5520	ERR_5520	SFL_PARENT-sfl parent no defined.
0x5521	ERR_5521	SFL_ATTR-invalid subfile attribute.

SEE ALSO

DTTSUBFL - typedef definition in Ability typdef section.

REFERENCE NUMBER - 55

ADVANCED CONCEPTS

Adding to d-tree

This chapter lists the steps necessary to build on the base of definitions provided with d-tree.

9.1 Adding a New Ability

As you have already seen, d-tree is made up of a variety of abilities. d-tree was designed to allow the user to define their own abilities. There are a number of steps necessary in order to add a new ability to d-tree. In this discussion we will state each necessary step. Each step will then be illustrated , as we actually add a new ability to d-tree.

- **1) The Idea -** First a need (or an ability) must be conceptualized by the developer. In other words we must start with an idea for an ability that we want to add to d-tree. For the illustration, let's add an ability for "communications". Here's the conceptual view of our need (or ability): We would like to have a function we can call from our programs to perform communications with another computer. The definitions of where, how, and what to communicate need to be defined as a "generic" so that we can use this function in a variety of applications. Rather than "hard coding" any specifics to the communication function, we will store the "specifics" in a structure. The communication function will be passed a pointer to this structure, where it can get the requirements to perform that specific communication. By simply changing the data in the structure, we can change the communications definition. In order to initialize this structure easily with "our specific" comminication information, we will use a "script" interface. The script provides an easy manner by which the user can define the communications for a specific application. Parsing this script will initialize the "communications structure". With this definition in memory, the "communications function" can be called to perform the task. The following steps will pull this all together.

- **2) Reference Word -** For coding consistency, we assign a five (5) letter reference word to our ability. In this case we'll use "**COMUN**". This is used as we assign variable names and definition names to d-tree. (i.e. IMAGE, FIELD, SCANN, and now COMUN)

- **3) "dt_defin.h"** - Next we must assign a reference number for our ability to d-tree. This is done by going into the file "dt_defin.h". Look for the #define DTKLAST. Insert a line above the #define DTKLAST assigning your new ability a reference number. The #define for your new ability should be named DTK????? where ????? is the reference word defined in step 2. In our case it will be **DTKCOMUN**. The number we assign to our DTK????? definition should be the number now defined for DTKLAST. Once we have done this we must increment DTKLAST by one (1). See illustration below:

```
                              dt_defin.h
#define    DTZHELPP 27  /* define HELPP Text reference number    */
#define    DTKMENUS 28  /* define MENUS keyword reference number */
#define    DTZMENUS 29  /* define MENUS textreference number */
#define    DTKRTREE 30  /* define RTREE keyword reference number */
#define    DTZRTREE 31  /* define RTREE text reference number */
#define    DTKTABLE 32  /* define TABLE keyword reference number */
#define    DTZTABLE 33  /* define TABLE text reference number */
#define    DTKHOOKS 34  /* define HOOKS keyword reference number */
#define    DTZHOOKS 35  /* define HOOKS text reference number */
#define    DTKMAPIT 36  /* define MAPIT keyword reference number */
#define    DTKTEMPP 37  /* define TEMPP keyword reference number */
#define  ->DTKCOMUN 38  /* this is our new communications ability */
#define    DTKLAST  39<-/* Last Ability number */

          Add New reference.
          Increment DTKLAST.
```

- **4) "dt_typdf.h"** - Next we need to define the structure to store the definition elements for our ability. We actually create a typdef of "structure type". This allows us to allocate a block of memory of this "structure type". (remember an ADAM in section 4). In our case let's assume our communication function needs the following data to perform its task. This information will be provided by the developer from the "script".

  a) Communication port id to use for communications.
  b) Modem Dial Command.
  c) Phone number to dial.
  d) login.
  e) password.
  f) file name containing list of data files to transmit/or receive.
  g) Modem Hang Up Command.

Insert your defined structure anywhere within the existing ability definitions in the file "dt_typdf.h". The other ability definitions should be obvious in the file. Each starts with a **#ifdef DTK?????** followed by its specific typedef. Note the naming convention for ability typdefs. Each

is defined as DTT????? where the ????? is the reference word from step 2. Our structure type is **DTTCOMUN**. The following shows our structure:

```
dt_typdf.h
/*****************************************************************************/
/* COMUN definitions */
#ifdef DTKCOMUN
typedef struct {
COUNT    num;         /* communication definition number */
COUNT    port;        /* Communication port id to use for comminications. */
TEXT     *dial;       /* Moden Dial Command.    */
TEXT     *phone;      /* Phone number to dial. */
TEXT     *login;      /* login.                 */
TEXT     *passwd;     /* password.              */
TEXT     *filelist;   /* file containing data files to transmit/or receive. */
TEXT     *hangup;     /* Modem Hang Up Command.    */
} DTTCOMUN;

#endif
/*****************************************************************************/
```

- **5) "dt_defin.h"** - As you can see, we have defined text pointers in our structure. This implies that we will have text stored as strings in memory that is referenced by the pointers defined in our structure. In order for d-tree to support the memory required to store these strings we must go back and add another reference definition in "dt_defin.h". THIS IS ONLY NECESSARY FOR ABILITIES WHOSE STRUCTURE DEFINITIONS REFER TO STRING OF TEXT THRU CHARACTER POINTERS. Add another #define for this ability. This #define should immediately follow the #define DTK????? that we inserted in step 3. It should have the same name except instead of DTK????? make it **DTZ?????**. The number assigned to this #define must be the DTK????? + 1. Remember to increase the #define DTKLAST by one (1). DTKLAST must always be one greater than the last number assigned. Our example would look as follows:

```
dt_defin.h
#define    DTZHELPP 27  /* define HELPP Text reference number    */
#define    DTKMENUS 28  /* define MENUS keyword reference number */
#define    DTZMENUS 29  /* define MENUS textreference number */
#define    DTKRTREE 30  /* define RTREE keyword reference number */
#define    DTZRTREE 31  /* define RTREE text reference number */
#define    DTKTABLE 32  /* define TABLE keyword reference number */
#define    DTZTABLE 33  /* define TABLE text reference number */
#define    DTKHOOKS 34  /* define HOOKS keyword reference number */
#define    DTZHOOKS 35  /* define HOOKS text reference number */
#define    DTKMAPIT 36  /* define MAPIT keyword reference number */
#define    DTKTEMPP 37  /* define TEMPP keyword reference number */
#define    DTKCOMUN 38  /* this is our new communications ability */
#define   ->DTZCOMUN 39  /* this is for the text needed by our new ability */
#define    DTKLAST  40<-/* Last Ability number */

              Added New Reference for Text.
              Incremented DTKLAST.
```

- **6) "dt_valid.h"** - The next step is to assign a keyword for our ability which will be recognized at parse time. We must also tell d-tree the name of a "parsing routine" to call when it finds a section of a d-tree script which defined this ability. Parsing routines follow the naming convention DTP????? (we will write our parsing routine for our ability in steps below). First add the declaration of the parsing routine as shown in the following illustration. Then add an entry into the table DTTKEYWD dt_kwd[]. Each table entry contains three fields:

  1) the keyword used in the script to identify this ability.
  2) the name of the parsing routine for this ability (DTP?????).
  3) the reference id for this ability as defined in step 3 (DTK?????)
     (note the DTZ????? never applies here).

dt_valid.h

```
#ifdef     DTKHOOKS        /* if HOOKS  */
COUNT      DTPHOOKS();     /* HOOKS Parsing Function */
#endif

#ifdef     DTKCOMUN        /* if COMMUNICATIONS  */
COUNT      DTPCOMUN();     /* COMUN Parsing Function */
#endif

/************************************/
/* Valid d-tree Script Keywords */
#ifndef IC

DTTKEYWD dt_kwd[] = {
/* KEYWORD, PARSING FUNCTION,REF NO                    */

#ifdef DTKCOMUN
{ "COMMUNICATIONS", DTPCOMUN  ,   DTKCOMUN } ,  /* FIELD   keyword */
#endif

#ifdef DTKFIELD
{ "FIELD",        DTPFIELD    ,   DTKFIELD } ,  /* FIELD   keyword */
#endif
```

Add your parsing declaration here

Add your table entry here. Keyword, parsing function, ref.

- **7) "dt_globl.h"** - next we must define our ability in a global array used by d-tree to manage the ability memory blocks. This array is simply a table of ability structure sizes. We must add both our DTK????? and DTZ????? (remember DTZ????? only applies to abilities that require text strings). The size of our DTK????? is simply defined by the size of the define typedef. (i.e. sizeof(DTTCOMUN)). DTZ????? sizes are always defined as sizeof(TEXT) for all abilities. Note the definition of this array:

  UCOUNT DTGSIZOF[DTKLAST] = { /* size of each element*/

This is an array indexed by the ability reference id (#define DTK????? or #define DTZ?????). **We must ensure that our entry into this table is placed in the proper occurance of the array**. In other words if our ability is defined as #define DTKCOLUM 38, our entry for this ability must be in the 38 (remember the array starts with the zero (0) occurance of the array). This is why ability reference numbers defined in "dt_defin.h" must start with zero (0) and be numbered consecutively. The addition for our new ability is show below:

dt_globl.h

```
UCOUNT  DTGSIZOF[DTKLAST] = {          /* size of each element */
sizeof(DATOBJ),         /*  DTKDDODA  */
sizeof(IFIL),           /*  DTKIFILS  */
sizeof(IIDX),           /*  DTKIIDXS  */
sizeof(ISEG),           /*  DTKISEGS  */
sizeof(DTTKEYBD),       /*  DTKKEYBD  */
sizeof(DTTFUNCT),       /*  DTKFUNCT  */
sizeof(DTTGROUP),       /*  DTKGROUP  */
sizeof(DTTRELAT),       /*  DTKRELAT  */
sizeof(DTTGENRL),       /*  DTKGENRL  */
sizeof(TEXT),           /*  DTZGENRL  */
sizeof(DTTIMAGE),       /*  DTKIMAGE  */
sizeof(DTTFIELD),       /*  DTKFIELD  */
sizeof(TEXT),           /*  DTZFIELD  */
sizeof(DTTCONST),       /*  DTKCONST  */
sizeof(TEXT),           /*  DTZCONST  */
sizeof(DTTPRMPT),       /*  DTKPRMPT  */
sizeof(TEXT),           /*  DTZPRMPT  */
sizeof(DTTSCANN),       /*  DTKSCANN  */
sizeof(DTTSUBFL),       /*  DTKSUBFL  */
sizeof(TEXT),           /*  DTZSUBFL  */
sizeof(DTTEDITS),       /*  DTKEDITS  */

sizeof(TEXT),           /*  DTZEDITS  */
sizeof(DTTDFALT),       /*  DTKDFALT  */
sizeof(TEXT),           /*  DTZDFALT  */
sizeof(DTTCALCS),       /*  DTKCALCS  */
sizeof(TEXT),           /*  DTZCALCS  */
sizeof(DTTHELPP),       /*  DTKHELPP  */
sizeof(TEXT),           /*  DTZHELPP  */
sizeof(DTTMENUS),       /*  DTKMENUS  */
sizeof(TEXT),           /*  DTZMENUS  */
sizeof(DTTRTREE),       /*  DTKRTREE  */
sizeof(TEXT),           /*  DTZRTREE  */
sizeof(DTTTABLE),       /*  DTKTABLE  */
sizeof(TEXT),           /*  DTZTABLE  */
sizeof(DTTHOOKS),       /*  DTKHOOKS  */
sizeof(TEXT),           /*  DTZHOOKS  */
0,                      /*  DTKMAPIT  */
0,                      /*  DTKTEMPP  */
sizeof(DTTCOMUN),       /*  DTKCOMUN  */  <-+-- Add your entry here.
sizeof(TEXT),           /*  DTZCOMUN  */  <-+

};
```

- **8)** "**dt_compl.c**" - as illustrated in section four, the **dt_compl** function in d-tree will "dump" the parsed in ability definitions to disk, so that these definitions can be included in at compile time, thus eliminating the need for a parse. We must add our new ability to this function so it will handle it as well.

  a) First add the reference word define in step 2 to the table at the top of the file named

  TEXT *DTSNAMES[DTKLAST] = {

  This table follows the same occurance rules as defined for the table in step 7. All DTZ????? definition occurance are defined in this table as "". The following shows our new entry:

```
dt_compl.h
TEXT *DTSNAMES[DTKLAST] = {
"DDODA",   /* DTKDDODA  */
"IFILS",   /* DTKIFILS  */
"IIDXS",   /* DTKIIDXS  */
"ISEGS",   /* DTKISEGS  */
"KEYBD",   /* DTKKEYBD  */
"FUNCT",   /* DTKFUNCT  */
"GROUP",   /* DTKGROUP  */
"RELAT",   /* DTKRELAT  */
"GENRL",   /* DTKGENRL  */
"",        /* DTZGENRL  */
"IMAGE",   /* DTKIMAGE  */
"FIELD",   /* DTKFIELD  */
"",        /* DTZFIELD  */
"CONST",   /* DTKCONST  */
"",        /* DTZCONST  */
"PRMPT",   /* DTKPRMPT  */
"",        /* DTZPRMPT  */
"SCANN",   /* DTKSCANN  */
"SUBFL",   /* DTKSUBFL  */
"",        /* DTZSUBFL  */
"EDITS",   /* DTKEDITS  */
"",        /* DTZEDITS  */
"DFALT",   /* DTKDFALT  */
"",        /* DTZDFALT  */
"CALCS",   /* DTKCALCS  */
"",        /* DTZCALCS  */
"HELPP",   /* DTKHELPP  */
"",        /* DTZHELPP  */
"MENUS",   /* DTKMENUS  */
"",        /* DTZMENUS  */
"RTREE",   /* DTKRTREE  */
"",        /* DTZRTREE  */
"TABLE",   /* DTKTABLE  */
"",        /* DTZTABLE  */
"HOOKS",   /* DTKHOOKS  */
"",        /* DTZHOOKS  */
"MAPIT",   /* DTKMAPIT  */
"",        /* DTKTEMPP  */
"COMUN",   /* DTKCOMUN  */
"",        /* DTZCOMUN  */      Add new entry here.
};
```

b) next we need to define a local work pointer in the function **DT_COMPL** of our new ability type. Our new ability pointer definition is shown below:

dt_compl.c

```
COUNT DT_COMPL(filename)
TEXT filename[];                          /* file name to write definitions */
{
COUNT DT_CKHRD(); /* check to see if ability is already hard coded */
COUNT DT_COMPI();
VOID  vtclose();
FILE  *fopen(), *fp;
TEXT  wrkfld[DT_FLDLN];
TEXT  wrkfld2[DT_FLDLN];
TEXT  *wrkptr;
COUNT linktyp=1;  /* type of source file to create */
COUNT useit;

FAST COUNT c,cc;
DTTFUNCT *funcptr;

DTTGENRL *gptr;
DTTRELAT *rptr;

#ifdef    DTKCOMUN
DTTCOMUN *comunptr;
#endif
```

Add your work pointer here.

c) Use another ability as a guide (copy another and change it) to write a loop that will write your structure to disk. Use the same loop as any other ability, but change the reference word (?????) to your ability reference word. Here is our new ability:

dt_compl.c

```
/************************************************************************************/
#ifdef    DTKCOMUN
if (DTGNUMBR[DTGCURGP][DTKCOMUN] && !DT_CKHRD(DTKCOMUN))
 {

  fprintf(fp,"DTTCOMUN DTSCOMUN[] = { /* COMUN */\n");

  pptr=((DTTCOMUN *)DTGPOINT[DTGCURGP][DTKCOMUN]);
  for (c=0; c<DTGNUMBR[DTGCURGP][DTKCOMUN]; ++c)
   {
    fprintf(fp,"{ ");  /* beginning of record */

    fprintf(fp,"%d,%d,",
    comunptr->num;         /* communication definition number */
    comunptr->port);       /* Communication port id to use for comminications.

    if (comunptr->dial)
    fprintf(fp,"\"%s\" ",pptr->dial);       /* Moden Dial Command.    */
    else
    fprintf(fp,"\"\" ");

    if (comunptr->phone)
```

```
    fprintf(fp,"\"%s\" ",pptr->phone);         /* Phone number to dial. */
    else
    fprintf(fp,"\"\" ");

    if (comunptr->login)
    fprintf(fp,"\"%s\" ",pptr->login);         /* login.                */
    else
    fprintf(fp,"\"\" ");

    if (comunptr->passwd)
    fprintf(fp,"\"%s\" ",pptr->passwd);         /* password.             */
    else
    fprintf(fp,"\"\" ");

    if (comunptr->filelist)
    fprintf(fp,"\"%s\" ",pptr->filelist);       /* file containing data files *
    else
    fprintf(fp,"\"\" ");

    if (comunptr->hangup)
    fprintf(fp,"\"%s\" ",pptr->hangup);        /* Modem Hang Up Command.   */
    else
    fprintf(fp,"\"\" ");

    fprintf(fp," } ,\n"); /* end of record */

    ++comunptr;
   }
  fprintf(fp,"};\n\n");  /* end structure */
 } /* end COMUN */
#endif
/*****************************************************************************/
```

- 9) The next step is the most difficult part of adding an ability. It is up to the user to define the syntax within the script for the new ability, and to write a parsing routine that will initialize the ability structures with the definition from the script. When the primary parsing routine detects the ability keyword, it will build a work buffer with just that ability's section of the script. The address and the length of this work buffer is then passed to the parsing routine that was defined in the table defined in step 6. This parsing routine must be defined as receiving the following parameters:

```
COUNT DTP?????(kwptr,parsebuf,len)/* ?????  Parsing Routine*/
DTTKEYWD  *kwptr;              /* ptrs to keyword structure */
FAST TEXT *parsebuf;     /* address of buffer with definition */
COUNT len;               /* length of buffer */
```

- There are plenty of examples of parsing routines within d-tree, as every ability has one. They all follow the naming convention DTP?????. below are listed some of the routines d-tree provides to aid in writing a parsing routine:

  DT_KEYNM - Validate token as a valid key name.
  DT_TOKNX - get next token from the temporary buffer passed to the parsing routine.
  DT_GENRL - Validate Token in a General Table.
  DT_TDODA - validate token as valid doda fiel symbol name.
  DT_CPMEM - reallocate ability memory.

  There is a skeleton parsing routine in the file "**dtpstart.c**" that may be used as a "starter" for your own parsing routine. This file is shown below with conceptual comments on the general necessities of a parse:

```
dtpstart.c
#include "dt_defin.h"   <----------   Always define d-tree's header.

/*************************************/
/* Valid symbols */
DTTGENRL dt_genedt[] = {    <---------
{"MANDATORY"   ,DTETMAND }, /* mandatory entry */
{"MAND_FILL"   ,DTETFILL }, /* mandatory fill */
{"",-1}                     /* terminaton Indicator */
};
/*************************************/

    If there are certain tokens defined in your syntax for special
    meanings they may be defined in a DTTGENRL type table at the top of
    your parseing routine. The function DT_GENRL can be used to determin
    if a token in in this table. Example of EDITS table.

/*********************************************************************/
/*                              DTP?????                             */
COUNT DTP?????(kuptr,parsebuf,len)   /* ????? Parsing Routine         */
DTTKEYWD  *kuptr;                    /* ptrs to keyword structure     */
FAST TEXT *parsebuf;                 /* address of buffer with definition */
COUNT len;                           /* length of buffer */
{
COUNT     atoi();
TEXT      *strbuf,*endbuf;      /* start end end buffer pointers      */
COUNT     notused;              /* work flags                         */
DATOBJ    *DT_TDODA();          /* validate token as valid doda symbol */
COUNT     DT_CPMEM();           /* reallocate ability memory          */
COUNT     DT_KEYNM();           /* validate token as valid key symbol */
COUNT     DT_TOKNX();           /* get next token from parsing buffer */
COUNT     DT_SPTRS();           /* reset relate pointers              */
DTTGENRL  *DT_GENRL(),*gptr;    /* general table validate             */
DTT?????  *???ptr;              /* work pointer to ????? structure    */
DTTRELAT  *rptr,*rptr2;         /* work pointer to relate structure   */
DATOBJ    *dodaptr;             /* work pointer to the doda           */
TEXT      *???txt;              /* pointer to ????? text              */
TEXT      token[DT_TOKLN];      /* token work buffer                  */
COUNT     c,ccc;                /* work fields                        */
```

```
COUNT     section:                  /* parsing stage indicator */
COUNT     ?????no:                  /* ????? number work field */
COUNT     noof????? = 0:            /* number of ????? */
COUNT     nooofflds  = 0:           /* number of fields involved */
COUNT     textlen   = 0:            /* total length of all text */
COUNT     reset     = 0:            /* need to reset RELAT pointers  */
COUNT     length:                   /* token length */
```

Here we have defined many common work variables that can may be needed in a typical parsing routine.

```
uerr_cod=0;
strbuf=parsebuf;
endbuf=strbuf+len-1;

/* first count the number of ??? by  counting ??? */
while ((length=DT_TOKNX(token,&strbuf,&endbuf,
                       &notused,&notused,&notused,&notused,&notused)))
 {
  if (DT_KEYNM(token)) continue; /* bypass key names */
  if (DT_TDODA(((DATOBJ *)DTGPOINT[DTGCURGP][DTKDDODA]),token))
    { ++noofflds; continue; } /* count fields */
  if ((gptr=DT_GENRL(dt_yours,token,0))) { ++noof?????; continue; }
  textlen+=(length+1); /* add up text length-add one for space or null*/
 }

/* reset buffer pointers */
strbuf=parsebuf;
endbuf=strbuf+len-1;
```

The first thing that must be done is to count how much allocated space is necessary for this ability. We need to determin both how many structures need to be allocated as well as how much text space is needed.

```
/*****************************************************************/
/* Allocate ????? Space */
if (noof?????)
{
if (DTGNUMBR[DTGCURGP][DTK?????]) reset=1;
if (DT_CPMEM(&DTGNUMBR[DTGCURGP][DTK?????],noof?????,
   (TEXT **) &DTGPOINT[DTGCURGP][DTK?????],sizeof(DTT?????)))
     return(ERR_6601);
DTGSIZOF[DTK?????]=sizeof(DTT?????);

/* Allocate ????? Text Space */
???txt=DTGPOINT[DTGCURGP][DTZ?????]; /* save old global ptr */
if ( DT_CPMEM(&DTGNUMBR[DTGCURGP][DTZ?????],textlen,
     (TEXT **)&DTGPOINT[DTGCURGP][DTZ?????],sizeof(TEXT)))
              return(ERR_6602);
DTGSIZOF[DTZ?????]=sizeof(TEXT);
```

Here we allocated the space, first for the structures, and then for the text.

```
/* now loop thru ????? defs and reset ptrs */
/* new ptr = new global plus (old address minus saved global) */
???ptr=((DTT????? *)DTGPOINT[DTGCURGP][DTK?????]);
for (c=0; c<(DTGNUMBR[DTGCURGP][DTK?????]-noof?????); ++c)
      {
       if (???ptr->edttxt)
       { ccc=???ptr->edttxt-???txt;
         ???ptr->edttxt=DTGPOINT[DTGCURGP][DTZ?????]+ccc; }
       if (???ptr->addtxt)
       { ccc=???ptr->addtxt-???txt;
         ???ptr->addtxt=DTGPOINT[DTGCURGP][DTZ?????]+ccc; }
       ++???ptr;
      }
???ptr=((DTT????? *)DTGPOINT[DTGCURGP][DTK?????]);
???ptr+=(DTGNUMBR[DTGCURGP][DTK?????]-noof?????);

???txt=DTGPOINT[DTGCURGP][DTZ?????];
???txt+=(DTGNUMBR[DTGCURGP][DTZ?????]-textlen);
} /* end if ????? */
```

We need to loop thru and reset any text pointers for any previously parsed ability of this same type.

```
?????no=kuptr->ocur[0];  /* set ????? no  */

if (kuptr->ocur[1])
 {
  kuptr->ocur[0]=kuptr->ocur[1];
  kuptr->ocur[1]=0;
 }
 else
  kuptr->ocur[0]=0;
```

All parsing routines need to contain this logic for ability reference numbers.

```
section=0; /* work field */
while ((length=DT_TOKNX(token,&strbuf,&endbuf,
                        &notused,&notused,&notused,&notused,&notused)))
 {
 switch (section)
  {
   case 0: /* example switch */
           if (!(gptr=DT_GENRL(dt_yours,token,0)))
                 { uerr_cod=ERR_100A; goto reterr; }
   ???ptr->field=your_structure_field;
  } /* end switch */
 }  /* end looping thru tokens */

return(0);

reterr:
printf("token='%s'\n",token);
return(uerr_cod);
} /* end of parseing routine */
```

Now write a loop that re-parses the same buffer. This time initilizing the allocated buffers.

- **10)** The last step is to write a function, that given a pointer to the structure will perform the appropriate task. Following d-tree's naming conventions, our new ability function whould be named DT_COMUN, although it doesn't make any difference. Our function would receive a pointer of our ability type:

  COUNT DT_COMUN(ptr) DTTCOMUN *ptr;

  Adding a new ability is a very powerful feature in d-tree. Steps 9 and 10 are the most difficult. The EDITS parsing function is a good function to study. It does not have a terribly complex syntax.

9.2 Adding a New FIELD Input Attribute.

In order to a a new input attribute for the FIELDS ability do the following steps:

- **1) "dt_typdf.h"** - Add a new #define reference for the new attribute in dt_typdf. Input attributes definitions begin with DTIA????.

```
                                    dt_typdf.h
/* input attributes */

#define DTIANONE     0X00   /* no attribute */
#define DTIANOCHG    0X01   /* no changes allowed via input */
#define DTIANUM      0X02   /* numbers only can be entered */
#define DTIACAPS     0X04   /* convert every char to caps */
#define DTIANMCAP    0X08   /* name input attribute */
#define DTIAFWCAP    0X10   /* first letter of first word of field to be caps */
#define DTIAAWCAP    0X20   /* first letter of all words in field to be caps */
#define DTIAPROCT    0X40   /* protected field-cursor will not enter */
#define DTIALOKUP    0X80   /* lookup detected */
#define DTIASCROL   0X100   /* scroll allowed */
#define DTIATABLE   0X200   /* TABLE_IN convertion */
#define DTIAHELPP   0X400   /* display HELP TEXT automatically */
#define DTIAGOTO    0X800   /* display GO TO another field */
#define DTIASP1    0X1000   /* special needs 1 */
#define DTIASP2    0X2000   /* special needs 2 */
#define DTIASP3    0X3000   /* special needs 3 */
#define DTIASP4    0X8000   /* spe
                                 <------ Add a new #define here.
```

- **2) "dtpfield.c"** - Add new attribute keyword for d-tree script in the input attribute table at the top of "dtpfield.c"

```
                                    dtpfield.c
/***************************************/
/* Valid INPUT Arributes */
DTTGENRL dt_iatr[] = {
{"NONE"         , DTIANONE  },
{"NOCHANGE"     , DTIANOCHG },
{"NUMERIC"      , DTIANUM   },
{"ALLCAPS"      , DTIACAPS  },
{"NAMECAPS"     , DTIANMCAP },
{"FRSWORDCAPS"  , DTIAFWCAP },
{"ALLWORDCAPS"  , DTIAAWCAP },
{"PROTECT"      , DTIAPROCT },
{"SCROLL"       , DTIASCROL },
{"TABLE_IN"     , DTIATABLE },
{"AUTO_HELP"    , DTIAHELPP },
{"GOTO"         , DTIAGOTO  },
{"NONE_NONE"    , DTIASP1   },
{"NAME_NONE"    , DTIASP2   },     Add your new input attribute here.
{"NAME_NUM"     , DTIASP3   },
{"NONE_NUM"     , DTIASP4   },
                             <----
{"",-1}                     /* terminaton Indicator */
};
```

- **3)** Logic must be added to the appropriate function to address this new attribute. Most input attributes are handled in d-tree by the low level input routine "dt_input" in file "dt_input.c". Some are addressed before the input routine is even called, in the "field out" routine (DT_FLDOT in file "dt_field.c").

9.3 Adding a New FIELD Output Attribute.

Output attributes fall into two categories: attributes that require no special screen control, and attributes that do require special screen control. In order to a new output attribute for the FIELDS ability do the following steps:

- 1) **"dt_typdf.h"** - Add a new #define reference for the new attribute in file "dt_typdf.h". Output attributes are defined at the bottom of this file. If special screen control (escape sequences) are necessary to make sure the #define DT_MXSEQ is set to handle enough sequences.

#define DT_MXSEQ 6 /* Maximum characters in keyboard special sequence */

dt_typdf.h

```
/* OUTPUT ATTRIBUTES                                                         */
/* note-do not assign (-1) to any screen seq or keyboard key                 */
/*      for (-1) is used to detect termination                               */
/* remember that all screen sequence numbers must be numbered                */
/* sequentually without skipping any numbers between the first               */
/* and last number                                                           */
/* there are two types of output attributes for a field                      */
/* screen control seq are one type, and simple output attributes that        */
/* do not involve special screen seq are another                             */

/* regular output attributes -no screen special seq                          */

#define DTOAGIDLIN '_'  /* input guide line character                        */
#define DTOANOLIN  (-2) /* do not display input guide lines                  */
#define DTOATABLE  (-3) /* TABLE_OUT convertion                              */
#define DTOAZEROS  (-4) /* Allow Zero to display (non-zero suppress)         */

/**********************/
/* screen special sequence output attributes                                 */
#define DTSCFRS     (-10)  /* define first screen sequence number */
#define DTSCLST     (-39)  /* define last  screen sequence number */
```

- 2) **"dtpfield.c"** - add the new attribute to the valid output attribute table in the field parsing function as shown below. The "dtpfield.c" module must then be recompiled.

dtpfield.c

```
/*************************************/
/* Valid OUTPUT Arributes */
DTTGENRL dt_oatr[] = {
{"RI"          ,DTSCRI },
{"INPUTRI"     ,DTSCLOTUS },        <---- Add your new attribute to this table.
{"UL"          ,DTSCUL },
{"HL"          ,DTSCHL },
{"EOL"         ,DTSCEOL },
{"NOLINES"     ,DTOANOLIN },
{"TABLE_OUT"   ,DTOATABLE },
{"ZERO"        ,DTOAZEROS },
{"BLACK"       ,DTSCBLACK },  /* black                  */
{"BLUE"        ,DTSCBLUE  },  /* blue                   */
{"GREEN"       ,DTSCGREEN },  /* green                  */
{"CYAN"        ,DTSCCYAN  },  /* cyan                   */
{"RED"         ,DTSCRED   },  /* red                    */
{"MAG"         ,DTSCMAG   },  /* magenta                */
```

- **3) "dt_keybd.h"** - If the output attribute definition is to come from the TERMCAP file (screen control sequence) add the attribute to the table shown below:

```
dt_keybd.h
/* Valid KEYBD types symbols */
DTTGENRL dt_genkey[] = {            <--- look at this table in dt_keybd.h
{"ESC",DTKBESC},
{"CR" ,DTKBCR},      /* Cr-Return on keyboard                */
{"BU" ,DTKBBU},      /* back-up on keyboard                  */
{"DC" ,DTKBDC},      /* delete character                     */
{"IC" ,DTKBIC},      /* insert char                          */
{"DW" ,DTKBDW},      /* down key                             */
{"UP" ,DTKBUP},      /* up key                               */
{"PD" ,DTKBPD},      /* page-down                            */
{"PU" ,DTKBPU},      /* page-up                              */
{"LF" ,DTKBLF},      /* left key                             */
{"RT" ,DTKBRT},      /* right key                            */
{"HM" ,DTKBHM},      /* home key                             */
{"EN" ,DTKBEN},      /* end key                              */
{"IL" ,DTKBIL},      /* Insert Line                          */
{"DL" ,DTKBDL},      /* Delete Line                          */
{"AD" ,DTKBAD},      /* Auto dup-Defaults                    */
{"PS" ,DTKBPS},      /* Print Screen                         */
{"F1" ,DTKBF1},      /* function key F1                 */
{"F2" ,DTKBF2},      /* function key F2                 */
{"F3" ,DTKBF3},      /* function key F3                 */
```

The parsing routine which reads the TERMCAP file must then be recompiled. Compile the module "dtpkeybd.c".

- **3) "dtermcap.h"** - If you would like this output attribute to be prompted in the dtermcap program enter the attribute in the following table and recompile the dtermcap program:

```
dtermcap.h
/****************************************/
/* Valid KEYBD types symbols */        <--- look at this table in dtermcap.h
DTTGENRL dt_gencap[] = {
{"Enter Terminal Name..........................",0},
{"Enter Escape Key.............................",DTKBESC},
{"Return Key...................................",DTKBCR},
{"Backup Key...................................",DTKBBU},
{"Delete Character.............................",DTKBDC},
{"Insert Character.............................",DTKBIC},
{"Down Key.....................................",DTKBDW},
{"Up Key.......................................",DTKBUP},
{"Page Down....................................",DTKBPD},
{"Page Up......................................",DTKBPU},
{"Left Key.....................................",DTKBLF},
{"Right Key....................................",DTKBRT},
{"Home Key.....................................",DTKBHM},
{"Post Key.....................................",DTKBEN},
{"Insert Line Key..............................",DTKBIL},
{"Delete Line Key..............................",DTKBDL},
{"Default Control Key..........................",DTKBAD},
{"Print Screen Key (optional-skip in DOS)......",DTKBPS},
{"Help Key.....................................",DTKBHELP},
```

- **4)** Screen control attributes will be handled by the DT_SCSEQ function in "dt_misci.c" and should require no more modifications. Other special purpose attributes may require changes to the code where output is done. See "dt_const.c" for constant out function and "dt_field.c" for field out function (DT_FLDOT and DT_FLDLO). Refer to the function reference section for function descriptions.

9.4 Adding a New EDIT.

In order to add a new field edit to d-tree do the following steps:

- **1) "dt_typdf.h"** - Add a new #define identifier for the edit in "dt_typdf.h" after the last edit defined. See below:

```
dt_typdf.h

/* edit types */
/* remember edit type numbers must start at one and be in seq order */
/* You MUST define DTETLAST to be same as last edit number          */

#define DTETMAND 1  /* MANDATORY edit.           */
#define DTETFILL 2  /* MANDATORY fill edit.      */
#define DTETDAT1 3  /* DATE edit.-MMDDYY         */
#define DTETDAT2 4  /* DATE edit.-MMYY           */
#define DTETDAT3 5  /* DATE edit.-MMDD           */
#define DTETTABL 6  /* TABLE edit.               */
#define DTETDUPK 7  /* Duplicate key edit.       */
#define DTETVALD 8  /* VALIDATE edit.            */
#define DTETSFLH 9  /* Subfile Hash edit.        */
#define DTETSFLS 10 /* Subfile SAME as edit.     */
#define DTETSFLN 11 /* Subfile NOT SAME as edit. */
#define DTETUSRE 12 /* User defined edit         */
                                                    Add your edit ID here.
#define DTETLAST 13 /* Last Edit Type Number     */
                                                    Increment this number.
```

- **2) "dt_edits.c"** - Write your own edit routine, and add the call to your edit routine to the switch in the DT_EDITS routine found in "dt_edits.c"

- **3) "dtpedits.c"** - Add your new edit reference with your function name to the valid edit table in the file "dtpedits.c".

```
dtpedits.c
DTTGENRL dt_genedt[DTETLAST+1] = {
{""             ,0        , ""                                        ,DT_RETRN},
{"MANDATORY"    ,DTETMAND , "DT_EMAND, /* mandatory entry        */",DT_EMAND},
{"MAND_FILL"    ,DTETFILL , "DT_EFILL, /* mandatory fill         */",DT_EFILL},
{"DATE_MMDDYY"  ,DTETDAT1 , "DT_EDATE, /* date edit-MMDDYY       */",DT_EDATE},
{"DATE_MMYY"    ,DTETDAT2 , "DT_EDATE, /* date edit-MMYY         */",DT_EDATE},
{"DATE_MMDD"    ,DTETDAT3 , "DT_EDATE, /* date edit-MMDD         */",DT_EDATE},
{"TABLE"        ,DTETTABL , "DT_ETABL, /* table validate         */",DT_ETABL},
{"DUPKEY"       ,DTETDUPK , "DT_EDUPK, /* duplicate keys         */",DT_EDUPK},
{"VALIDATE"     ,DTETVALD , "DT_EVALD, /* validate edit          */",DT_EVALD},
{"SFLHASH"      ,DTETSFLH , "DT_EDSFL, /* subfile hash           */",DT_EDSFL},
{"SFLSAME"      ,DTETSFLS , "DT_EDSFL, /* subfile same           */",DT_EDSFL},
{"SFLNOTSAME"   ,DTETSFLN , "DT_EDSFL, /* subfile not same       */",DT_EDSFL},
{"USER_EDIT"    ,DTETUSRE , "DT_RETRN, /* user defined edit      */",DT_RETRN},

{""             ,-1       , ""                                        ,DT_RETRN}
};
/*****************************************/

                  Add your new edit reference here.
```

- 4) Recompile "dtpedits.c" and "dt_edits.c".

APPENDIX A
Function Reference

* REFERENCE NUMBERS-

Function Name	Function Number	Description
DT_ADREC	30	Add Record to ctree file.
DT_ALDOD	1C	Doda Alignment. Given a DODA pointer, determain offsets and addresses.
DT_ALIGN	68	Set or verify addresses in DODA and determain machine alignments.
DT_AVREC	114	Add a Variable Length Record.
DT_BHELP	7E	Build help text index file.
DT_CALCS	83	CALCS processing function.
DT_CALMP	109	Perform field mapping with a calculation.
DT_CKHRD	115	Check to see if an ability is hard coded.
DT_CLEAR	23	Clear the Screen.
DT_CLEOL	50	Clear to the end of a line.
DT_CLRBK	3F	Clear a portion of the screen.
DT_CLRLN	116	Clear the screen from line number x to line number y.
DT_CMDOD	110	Compare two dodas and set mapping flag for file conversion. Used by DT_REFMT function. (ie: change fields, record length)
DT_COMPI	1F	Convert Allocated Typdef structures to compile time initilizion C code-Incremental file struct.
DT_COMPL	54	Convert Allocated Typdef structures to compile time initilizion C code.
DT_CONST	20	Display Constant (Contant out).
DT_CPMEM	2C	Ability Memory Allocation Routine.
DT_CVDOD	117	Convert data as define in one doda into the format defined by another doda.
DT_DELET	37	Delete characters from a string.
DT_DFALT	78	Default logic-field level.
DT_DFIMG	92	Default logic-IMAGE level.
DT_DFINI	6D	Execute Default Definitions
DT_DLREC	2D	Delete a c-tree record.
DT_DODBK	7B	Check to see of a doda field address if blank or zero
DT_DODTS	89	Create default d-tree script.
DT_DOINT	34	Initilize c-tree file record buffers.
DT_DOMSG	91	Display a message on the default message line.
DT_DOPAD	35	Pad fields in c-tree file record buffer.

DT_DORTS	90	Create default r-tree script
DT_DTICP	118	"In Comming priority" for DT_PSFIX function.
DT_DTISP	111	"In stack priority" for DT_PSFIX function.
DT_EDATE	2E	EDIT routine-Dates.
DT_EDITS	67	Primary field edit function.
DT_EDRRD	119	Read Read Record edit to prevent muliuser interferance.
DT_EDSFL	3B	Edit a subfile.
DT_EDUPK	2F	EDIT routine-Duplicate Keys.
DT_EFILL	3D	EDIT routine-Maditory Fill.
DT_EMAND	3A	EDIT routine-Mandatory Field.
DT_EQREC	4A	Equal Record function.
DT_ETABL	3E	EDIT routine-Table Edit.
DT_EVALD	3C	EDIT routine-Validate with another file.
DT_EVALU	6B	Evaluate a postfix expression.
DT_FLATI	103	Import a Flat File to a c-tree file
DT_FLATO	104	Output a data file to a Flat File.
DT_FLDIN	25	Input a field (Field In).
DT_FLDLO	7A	Field out-low level
DT_FLDNM	70	Validate token as valid field symbolic name in DODA
DT_FLDOT	21	Display Field (Field Out).
DT_FLDTX	4B	Convert a Ascii Field to Valif Field Type.
DT_FRAME	4C	Draw a Frame on the screen.
DT_FREEE	2A	Free All Ability's Allocated Space
DT_FSREC	99	Get the first record in a file.
DT_FSSFL	120	Read the first subfile record.
DT_FUNCT	69	Validate token as valid user defined function.
DT_GENRL	18	Validate token to a GENRL type table.
DT_GHELP	7D	Get help text from a text file.
DT_GPINN	107	Read a group from Disk.
DT_GPOUT	105	Write a group to Disk.
DT_HELPP	85	HELPP routine.
DT_HOOKS	102	Hook function.
DT_IFILS	81	Open IFILS.
DT_IMAGE	27	IMAGE OUT then IMAGE IN. Combination DT_IMGOT then DT_IMGIN.
DT_IMGAL	28	IMAGE ALL-Display and Input a range of IMAGE's.
DT_IMGIN	26	Input an Image (Image In). Series of DT_FLDIN's related to the provided IMAGE.
DT_IMGLG	1A	Redisplay IMAGE's from log.
DT_IMGMV	47	Same as DT_IMAGE but allows user to change IMAGE coordinates.
DT_IMGOT	19	Display IMAGE (Image out). Series of DT_FLDOT & DT_CONST related to provided IMAGE.

DT_INAME	53	Return the associated IMAGE number for token.
DT_INBUF	36	Initilize a memory buffer to nulls.
DT_INDOD	112	Initilize all addresses defined in the DODA.
DT_INPUT	4D	Low Level Keyboard Input Routine.
DT_KEYBD	72	Keyboard Initilization routine. Load temporary parsing buffer with terminal definition fromTERMCAP file for the DTPKEYBD funtion to parse.
DT_KEYCK	121	Validate Token valid termcap token.
DT_KEYCV	129	Convert key number identifier to actual key number.
DT_KEYNM	42	Validate token as a key symbolic name for a index.
DT_LOCAT	24	Position Cursor on Screen.
DT_LOCPT	75	Locate cursor for printed output.
DT_MAPIT	96	MAPIT routine. Map one field to another.
DT_MENUS	5F	Menu Function
DT_MNUCK	130	Check menu condition definition for a match.
DT_MPDOD	122	Map data for the DT_REFMT function.
DT_MSKOT	123	Strip Mask Characters off input field.
DT_NSERT	79	Insert a character into a string.
DT_NXREC	93	Get next record.
DT_NXSFL	131	Read the next subfile record.
DT_OFSET	86	Calculate the offset of a DODA entry.
DT_OVLAP	124	Detect if one image or field overlays another on the screen.
DT_PARSE	11	Primary Parsing Function.
DT_PBUFF	14	Load temporary parsing buffer.
DT_PRMPT	40	Prompt routine. Provides key no, target, significant length for accessing a c-tree file.
DT_PSTFX	6A	Convert an expression in infix form to postfix.
DT_PVREC	94	Get previous record.
DT_RCDLN	87	Calculate record length from DODA definition.
DT_RDBUG	44	RELAT debug function-display RELAT structure.
DT_REFMT	6C	Reformat a c-tree file.
DT_RSORT	132	Sort the RELAT structure.
DT_RSORT &		Sort RELAT structure.
DT_CMPAR	43	Compare used by DT_RSORT to determain order.
DT_RTRE2	125	R-tree interface secondary substitution function.
DT_RTREE	6E	r-tree front end
DT_RWREC	4F	Rewrite a c-tree record.
DT_SCANN	41	Scan routine for c-tree data file.
DT_SCGET	49	Do x number of PREVKEY's and then read in record. Used By DT_SCANN for record selection.
DT_SCSEQ	76	Output Screen Control Sequence.

DT_SETTY	52	Set TTY line to get one character at a time.(Unix)
DT_SFCAD	1E	Subfile Add for parent and child
DT_SFCLD	1D	Load child subfile.
DT_SFHAD	61	Add subfile to disk-high level.
DT_SFHDL	60	Delete subfile from disk-high level.
DT_SFHLD	59	Load Subfile Routine-high level.
DT_SFLAD	58	Add subfile to disk-low level.
DT_SFLDL	57	Delete subfile from disk-low level.
DT_SFLLD	56	Load Subfile Routine-low level.
DT_SFLNW	127	Add a new record to a subfile.(extend the subfile-unlimited sfl only).
DT_SFLOT	62	Display subfile (SUBFL Out).
DT_SFLRM	133	Remove a subfile from disk (unlimited sfl)
DT_SFLRW	126	Rewrite a record back to a subfile.
DT_SPTRS	22	Set relationship pointers in RELAT structure.
DT_STALN	88	Initilize (set align) the alignment array.
DT_SUBFL	64	Maintain a Subfile. (Subfile input routine).
DT_TDODA	17	Validate token is a symbolic name in DODA.
DT_TODAY	5C	Get System Date
DT_TOKEN	12	Get next token from file.
DT_TOKKW	13	Validate token as valid ABILITY keyword.
DT_TOKNX	15	Get next token from memory buffer.
DT_TSPLT	65	Token Split function. Convert a token in the form of XXXXXX(yyy) into yyy and XXXXXX.
DT_UNHID	134	Display the hide screen buffer to the screen.
DT_UNPAD	5B	Unpad a record structure.
DT_UNPAK	33	Low level Unpack. Unpack one buffer into another.
DT_UNPOP	128	Un-pop (clear and refresh behind) a image from the screen.
DT_VLINN	32	Un-Pack Variable Length record.
DT_VLOUT	31	Pack Variable Length record.
DT_XTRCT	45	Extract a subset of relations from RELAT structure.
DTVIMAGE	48	Get IMAGE pointer given IMAGE number. Often used to validate a IMAGE number.
DTPCALCS	82	Parsing for CALCS keyword.
DTPCONST	46	Parse CONST keyword.
DTPDFALT	77	DFALT (default) parsing routine
DTPEDIT2	113	Edits Parsing sub-function to handle text strings.
DTPEDITS	66	Parsing function for EDITS.
DTPFIELD	10	Parse FIELD keyword.
DTPGROUP	108	Parse group definition.
DTPHELPP	84	Parsing function for HELPP keyword.
DTPHOOKS	101	Parse CALCS Keyword.
DTPIFILS	80	Parsing function for IFILS.

DTPIMAGE	16	Parse IMAGE keyword.
DTPKEYBD	71	Parse KEYBD function. Used to parse the termcap definition.
DTPKEYST	74	Sort Keyboard array.
DTPMAPIT	95	Parse MAPIT keyword.
DTPMENUS	5E	Parsing Routine for MENUS
DTPPRMPT	38	Parse for PROMPT keyword.
DTPRTREE	6F	Parse r-tree interface definition
DTPSCANN	39	Parse for SCAN keyword.
DTPSUBFL	55	Parse for SUBFILE keyword.
DTPTABLE	7C	Parse TABLE keyword.
DTCATADI	A5	Ability Dictionary In/Delete routine
DTCATADO	A4	Ability Dictionary Out routine
DTCATCSP	A6	Create C language source specs
DTCATDIF	106	File difference routine
DTCATDOD	A3	Create doda stucture
DTCATINC	A1	Create Incremental Structures
DTCATPAR	A2	Create Paramter file

Last ref number used = 134

Note that error codes are hex in order to accomodate the first two digits to represent the module and the second two digits to be the error number within the module. Using hex allows more than 99 modules and more than 99 errors per module. Remember functions that return a value greater than 7F00 must be declared as UCOUNT.

APPENDIX B
ERROR MESSAGES

ERROR CODE	DESCRIPTION
• ERR_1001 0x1001	DTPFIELD-Symbolic Name Not Found In DODA
• ERR_1002 0x1002	DTPFIELD-Invalid Input Attribute
• ERR_1003 0x1003	DTPFIELD-Invalid Output Attribute
• ERR_1004 0x1004	DTPFIELD-Syntax error found
• ERR_1005 0x1005	DTPFIELD-Pointer not pointing to a valid field
• ERR_1006 0x1006	DTPFIELD-Could not allocate parse space
• ERR_1007 0x1007	DTPFIELD-No Image with same number found
• ERR_1008 0x1008	DTPFIELD-No Field to Image relationships
• ERR_1009 0x1009	DTPFIELD-Relationship pointer Incremented to Far
• ERR_100A 0x100A	DTPFIELD-Invalid user defined function or Invalid field symbol name
• ERR_100B 0x100B	DTPFIELD-Could not allocate memory for mask text
• ERR_100C 0x100C	DTPFIELD-Syntax error in mask definition
• ERR_100D 0x100D	DTPFIELD-Could not allocate space for doda symbol names
• ERR_1101 0x1101	DT could not open d-tree Script File
• ERR_1102 0x1102	DT no valid keyword found in script
• ERR_1103 0x1103	DT calloc for parsing buffer allocation
• ERR_1601 0x1601	DTPIMAGE-Could not Allocate Parse space
• ERR_1602 0x1602	DTPIMAGE-Invalid Optional Feature
• ERR_1603 0x1603	DTPIMAGE-Could not allocate space for DODA
• ERR_1604 0x1604	DTPIMAGE-Could not allocate space for DODA symbolic names text
• ERR_1605 0x1605	DTPIMAGE-INPUT_ADVANCE option invalid
• ERR_1606 0x1606	DTPIMAGE-There must be a space between input field and constant field
• ERR_1801 0x1801	DT_GENRL called with invalid table
• ERR_1802 0x1802	DT_GENRL could not allocate space for symbolic names
• ERR_1803 0x1803	DT_GENRL could not allocate space for symbolic name text
• ERR_1901 0x1901	DT_IMGOT-invalid image number
• ERR_1902 0x1902	DT_IMGOT-image has no fields
• ERR_1903 0x1903	DT_IMGOT-Could not open print file

- ERR_1904 0x1904 DT_IMGOT-Tried to Use TEMPP type that is already in use
- ERR_1905 0x1905 DT_IMGOT-Could not allocate TEMPP type space for print screen option
- ERR_2301 0x2301 DT_CLEAR-invalid screen coordinates
- ERR_2401 0x2401 DT_LOCAT-invalid screen coordinates
- ERR_2501 0x2501 DT_FLDIN-Input Longer than DT_FLDLN
- ERR_2601 0x2601 DT_IMGIN-invalid image number
- ERR_2602 0x2602 DT_IMGIN-image has no fields
- ERR_2901 0x2901 DT_INITL-First & last field name not found in DODA.
- ERR_3101 0x3101 DT_VLOUT-Could not allocate enough space for variable length packed buffer.
- ERR_3102 0x3102 DT_VLOUT-Could not find first variable length field in doda. Make sure that fixed record portion matches record length.
- ERR_3201 0x3201 DT_VLINN-REDVREC failed.
- ERR_3301 0x3301 DT_UNPAK-Could not find first variable length field in doda. Make sure that fixed record portion matchesrecord length.
- ERR_3701 0x3701 DT_DELET-delete position beyond end of string
- ERR_3702 0x3702 DT_DELET-triing to delete to many characters
- ERR_3801 0x3801 DTPPRMPT-Could not allocate space for prompt
- ERR_3802 0x3802 DTPPRMPT-Could not allocate space for relations
- ERR_3803 0x3803 DTPPRMPT-Invalid keyword-define USES_IMAGE(?)
- ERR_3804 0x3804 DTPPRMPT-Image number/name not defined correctly.
- ERR_3805 0x3805 DTPPRMPT-Key symbol name not in ISAM definition.
- ERR_3806 0x3806 DTPPRMPT-Field symbol name not found in DODA.
- ERR_3807 0x3807 DTPPRMPT-FIELD defined not on IMAGE defined.
- ERR_3808 0x3808 DTPPRMPT-Could not allocate space for prefix
- ERR_3809 0x3809 DTPPRMPT-Scann Number not defined
- ERR_380A 0x380A DTPPRMPT-Scann Number must be defined before target fields
- ERR_3901 0x3901 DTPSCANN-Could not allocate space for SCANN
- ERR_3902 0x3902 DTPSCANN-Invalid SCANN option defined
- ERR_3903 0x3903 DTPSCANN-IMAGE_OUT image no not a defined IMAGE.
- ERR_3904 0x3904 DTPSCANN-must have valid IMAGE_ROL imageno

• ERR_3905 0x3905	DTPSCANN-must have valid IMAGE_INP imageno
• ERR_4001 0x4001	DT_PRMPT-Prompt number not defined
• ERR_4002 0x4002	DT_PRMPT-Associated IMAGE not found
• ERR_4003 0x4003	DT_PRMPT-Prompt not found in extracted subset
• ERR_4004 0x4004	DT_PRMPT-Target fields not found for target build
• ERR_4005 0x4005	DT_PRMPT-Could not allocate space for extract
• ERR_4101 0x4101	DT_SCANN-scann number not a defined by SCAN
• ERR_4102 0x4102	DT_SCANN-IMAGE_OUT display failed
• ERR_4103 0x4103	DT_SCANN-EQLREC failed on previous displayed rcd.
• ERR_4501 0x4501	DT_XTRCT-Could not allocate extract space
• ERR_4601 0x4601	DTPCONST-Image number for constant not defined
• ERR_4602 0x4602	DTPCONST-Token not a valid constant or attribute
• ERR_4603 0x4603	DTPCONST-Invalid output attribute
• ERR_4604 0x4604	DTPCONST-Attribute does not refer to any constant
• ERR_4605 0x4605	DTPCONST-Could not allocate space for extract
• ERR_4701 0x4701	DT_IMGMV-imageno not a defined IMAGE.
• ERR_4901 0x4901	DT_SCGET-EQLREC FAILED.
• ERR_5401 0x5401	DT_COMPL-Could not rewrite text file
• ERR_5501 0x5501	DTPSUBFL-Could not allocate space for subfile definition
• ERR_5502 0x5502	DTPSUBFL-Could not allocate space for relationships
• ERR_5503 0x5503	DTPSUBFL-Parse is expecting to see a subfile keyword-syntax error
• ERR_5504 0x5504	DTPSUBFL-Parse cannot determain what sfl keyword is being parsed-syntax error
• ERR_5505 0x5505	DTPSUBFL-SFL_IMAGE number/name not defined correctly.
• ERR_5506 0x5506	DTPSUBFL-Invalid Number for MAX_RECORDS value.
• ERR_5507 0x5507	DTPSUBFL-Invalid Number for SFL_LINES value.
• ERR_5508 0x5508	DTPSUBFL-SFL_IMAGE must be define before SFL_TARGET
• ERR_5509 0x5509	DTPSUBFL-SFL_RECORDS must be define before SFL_TARGET
• ERR_5510 0x5510	DTPSUBFL-SFL_LINES must be define before SFL_TARGET

- ERR_5511 0x5511 DTPSUBFL-Key symbol name not in ISAM definition.
- ERR_5512 0x5512 DTPSUBFL-Syntax error-looking for Key symbol name
- ERR_5513 0x5513 DTPSUBFL-Field symbol name not found in doda.
- ERR_5514 0x5514 DTPSUBFL-Only one SFL_TARGET definition allowed
- ERR_5515 0x5515 DTPSUBFL-SFL_MAP Field symbol name not found in doda.
- ERR_5516 0x5516 DTPSUBFL-SFL_MAP length longer than child field
- ERR_5517 0x5517 DTPSUBFL-SFL_TARGET must define target field or prefix
- ERR_5518 0x5518 DTPSUBFL-SFL_MUSTHAVE field not found in DODA
- ERR_5519 0x5519 DTPSUBFL-SFL_BOUNDARY field not found in DODA
- ERR_5520 0x5520 DTPSUBFL-SFL_PARENT-subfile parent not defined
- ERR_5521 0x5521 DTPSUBFL-SFL_ATTR-invalid subfile attribute
- ERR_5601 0x5601 DT_SFLLD-Could not allocate space for subfile
- ERR_5602 0x5602 DT_SFLLD-Memory Block Option is Invalid
- ERR_5603 0x5603 DT_SFLLD-Could not Create Disk Subfile
- ERR_5801 0x5801 DT_SFLAD-Could not allocate space for extract
- ERR_5901 0x5901 DT_SFHLD-Invalid subfile number.
- ERR_5902 0x5902 DT_SFHLD-Could not allocate extract space
- ERR_5903 0x5903 DT_SFHLD-Could not allocate memory control block
- ERR_5904 0x5904 DT_SFHLD-Ocur Number out of range
- ERR_6001 0x6001 DT_SFHDL-Invalid subfile number.
- ERR_6002 0x6002 DT_SFHDL-Subfile not allocated
- ERR_6003 0x6003 DT_SFHDL-No of records loaded into subfile not same as the number that where deleted
- ERR_6101 0x6101 DT_SFHAD-Invalid subfile number.
- ERR_6102 0x6102 DT_SFHAD-Subfile not allocated
- ERR_6201 0x6201 DT_SFLOT-Invalid subfile number.
- ERR_6202 0x6202 DT_SFLOT-Subfile not allocated.
- ERR_6203 0x6203 DT_SFLOT-Must pass record number
- ERR_6204 0x6204 DT_SFLOT-Record number greater than max records
- ERR_6205 0x6205 DT_SFLOT-Imvalid image number for subfile
- ERR_6206 0x6206 DT_SFLOT-Child subfile number invalid
- ERR_6207 0x6207 DT_SFLOT-parent record number is child occurances
- ERR_6401 0x6401 DT_SUBFL-Invalid Subfile Number.

- ERR_6402 0x6402 DT_SUBFL-Subfile Not Allocated.
- ERR_6403 0x6403 DT_SUBFL-Invalid Image define for Subfile.
- ERR_6404 0x6404 DT_SUBFL-Could not allocate temporary save space
- ERR_6601 0x6601 DTPEDITS-Could not allocate EDITS structure
- ERR_6602 0x6602 DTPEDITS-Could not allocate memory for EDITS text
- ERR_6603 0x6603 DTPEDITS-Could not allocate EDITS's RELAT memory
- ERR_6604 0x6604 DTPEDITS-Edit type must follow edit text
- ERR_6605 0x6605 DTPEDITS-Edit Type must follow field name
- ERR_6606 0x6606 DTPEDITS-Must enter message text
- ERR_6607 0x6607 DTPEDITS-field not found on associated image
- ERR_6608 0x6608 DTPEDITS-DUPKEY edit must have key no or name
- ERR_6609 0x6609 DTPEDITS-DUPKEY key symbol length to short ex: key symbol= "ky" and the key number it represents is "100". The "ky" is only 2 digits, the "100" is 3 digits. Due to memory allocation this is invalid
- ERR_6701 0x6701 DT_EDITS-Memory Allocation error on extract
- ERR_6801 0x6801 DT_ALIGN-Calculated address not same as DODA
- ERR_6802 0x6802 DT_ALIGN-First & last field name not found in DODA.
- ERR_6803 0x6803 DT_ALIGN-Record length in IFIL not correct
- ERR_6804 0x6804 DT_ALIGN-Could Not Allocate Field Memory
- ERR_7101 0x7101 DTPKEYBD-Could not allocate space for key sequence
- ERR_7102 0x7102 DTPKEYBD-Syntax error in TERMCAP definition
- ERR_7103 0x7103 DTPKEYBD-DT_MXSEQ not large enough in dt_typdf.h
- ERR_7201 0x7201 DT_KEYBD-Could not open TERMCAP file
- ERR_7202 0x7202 DT_KEYBD-Terminal not found in TERMCAP file
- ERR_7203 0x7203 DT_KEYBD-Could not allocate temporary parsing space
- ERR_7701 0x7701 DTPDFALT-Could not allocate DFALT structure space
- ERR_7702 0x7702 DTPDFALT-Could not allocate memory for DFALT text
- ERR_7703 0x7703 DTPDFALT-Could not allocate DFALT's RELAT space
- ERR_7704 0x7704 DTPDFALT-Error-Default type must follow field name
- ERR_7705 0x7705 DTPDFALT-Stntax error-invalid field symbol name.
- ERR_7801 0x7801 DT_DFALT-Memory Allocation error on extract

- ERR_7901 0x7901 — DT_NSERT-Position to insert chararacter is passed end of string
- ERR_801 0x801 — DTPIFILS-Unable to allocate IFILS structures
- ERR_802 0x802 — DTPIFILS-Unable to allocate IIDXS structures
- ERR_803 0x803 — DTPIFILS-Unable to allocate ISEGS structures
- ERR_804 0x804 — DTPIFILS-Unable to allocate space for text information
- ERR_805 0x805 — DTPIFILS-Syntax error-Must have IFILS symbol
- ERR_806 0x806 — DTPIFILS-Field defined as key segment not in DODA
- ERR_807 0x807 — DTPIFILS-Field defined first or last field not in DODA
- ERR_808 0x808 — DTPIFILS-Field defined as first variable length field is not in DODA
- ERR_809 0x809 — DTPIFILS-Must define KEY_NAME before defining DUPS_OK
- ERR_811 0x811 — DT_IFILS-Invalid first field name
- ERR_812 0x812 — DT_IFILS-Invalid last field name
- ERR_813 0x813 — DT_IFILS-First and last field out of order
- ERR_814 0x814 — DT_IFILS-offset id for segment not between first and last fields of file
- ERR_8801 0x8801 — DT_STALN-COUNT, UCOUNT or POINTER are not correctly sized. Call Faircom
- ERR_8802 0x8802 — DT_STALN-This machine addresses 32 bit words (not bytes). Call FairCom
- ERR_8803 0x8803 — DT_STALN-This machine addresses words.(not bytes). Call FairCom
- ERR_891 0x891 — DT_DODTS-Could not open base script
- ERR_892 0x892 — DT_DODTS-Could not write temp dtree script
- ERR_893 0x893 — DT_DODTS-Could not open user script
- ERR_901 0x901 — DT_DORTS-Could not write new r-tree script
- ERR_902 0x902 — DT_DORTS-Could not open d-tree script
- ERR_903 0x903 — DT_DORTS-Could not find master screen
- ERR_921 0x921 — DT_DFIMG-Image number passed is not defined
- ERR_922 0x922 — DT_DFIMG-Could not allocate memory for extract
- ERR_951 0x951 — DTPMAPIT-Must have an even number of fields defined
- ERR_952 0x952 — DTPMAPIT-Could not allocate space for MAP relates
- ERR_953 0x953 — DTPMAPIT-Invalid Source Token-Not field name or CALCS name
- ERR_954 0x954 — DTPMAPIT-copy length longer than child field
- ERR_955 0x955 — DTPMAPIT-Must Define VALID Map Type
- ERR_956 0x956 — DTPMAPIT-destination field symbol name not found in doda.

- ERR_957 0x957 DTPMAPIT-destination field symbol name not on screen
- ERR_961 0x961 DT_MAPIT-Could not allocate space for extract
- ERR_971 0x971 DTPCREAT-Could not allocate space for IFIL
- ERR_972 0x972 DTPCREAT-Syntax error
- ERR_9801 0x9801 DTCOPCAT-Could not allocate space for CTFILE
- ERR_9802 0x9802 DTCOPCAT-Could not open catalog table file
- ERR_9803 0x9803 DTCOPCAT-Could not allocate space for IFIL
- ERR_9804 0x9804 DTCOPCAT-Could Not read definition record from catalog TABLE file
- ERR_9805 0x9805 DTCOPCAT-Could not open ifils
- ERR_9806 0x9806 DTCOPCAT-Could not find any TABLE column records
- ERR_9807 0x9807 DTCOPCAT-Could not allocate space for catalog DODA
- ERR_9808 0x9808 DTCOPCAT-Could not read catalog index definitions
- ERR_9809 0x9809 DTCOPCAT-Could not allocate space for IIDX
- ERR_9810 0x9810 DTCOPCAT-Could not allocate space for ISEG
- ERR_9811 0x9811 DTCOPCAT-Index column name not found in DODA
- ERR_1B01 0x1B01 DT_SFLNW-Could not allocate space for new subfile
- ERR_1D01 0x1D01 DT_SFCLD-Invalid Parent subfile
- ERR_1D02 0x1D02 DT_SFCLD-Invalid Child subfile
- ERR_1D03 0x1D03 DT_SFCLD-Parent not yet allocated
- ERR_1E01 0x1E01 DT_SFCAD-Invalid Parent subfile
- ERR_1E02 0x1E02 DT_SFCAD-Parent not yet allocated
- ERR_1E03 0x1E03 DT_SFCAD-Invalid child subfile
- ERR_1E04 0x1E04 DT_SFCAD-child not yet allocated
- ERR_A11 0xA11 DTCATINC-Unable to initlize display subfile
- ERR_A12 0xA12 DTCATINC-Display subfile number not defined
- ERR_A13 0xA13 DTCATINC-Invalid index subfile number
- ERR_A14 0xA14 DTCATINC-Invalid segment subfile number
- ERR_A15 0xA15 DTCATINC-Invalid fields or column subfile number
- ERR_A16 0xA16 DTCATINC-Could not allocate IFIL space
- ERR_A17 0xA17 DTCATINC-Could not allocate IIDX space
- ERR_A18 0xA18 DTCATINC-Could not allocate ISEG space
- ERR_A19 0xA19 DTCATINC-Could not allocate text space
- ERR_A1A 0xA1A DTCATINC-Could not open temp text work file
- ERR_A1B 0xA1B DTCATINC-DTCATDOD Error see uerr_cod

- ERR_A21 0xA21 DTCATPAR-Could not open temporary text work file
- ERR_A22 0xA22 DTCATPAR-Could not initialize work subfile.
- ERR_A23 0xA23 DTCATPAR-Work subfile not defined
- ERR_A24 0xA24 DTCATPAR-Could not find index subfile
- ERR_A25 0xA25 DTCATPAR-Could not find segment subfile
- ERR_A26 0xA26 DTCATPAR-Could not find column subfile
- ERR_A27 0xA27 DTCATPAR-DTCATDOD error see uerr_cod
- ERR_A31 0xA31 DTCATDOD-Invalid subfile number
- ERR_A32 0xA32 DTCATDOD-Could not allocate space for DODA
- ERR_A33 0xA33 DTCATDOD-Could not allocate space for DODA name txt
- ERR_A51 0xA51 DTCATADI-Program/Version not found in Program Dictionary
- ERR_A52 0xA52 DTCATADI-Program Has No Relationships in RELATE Dictionary
- ERR_A53 0xA53 DTCATADI-Ability record pointed to by entry from the relate dictionary not found
- ERR_A54 0xA54 DTCATADI-Could Not Allocate space for Abilitys
- ERR_A55 0xA55 DTCATADI-Invalid Ability type found in relate entry
- ERR_A56 0xA56 DTCATADI-Ability record pointed to by relate entry has zero length
- ERR_A57 0xA57 DTCATADI-Delete of Ability dictionary record failed-see uerr_cod
- ERR_A58 0xA58 DTCATADI-Delete of Relate Dictionary record failed see-uerr_cod
- ERR_A59 0xA59 DTCATADI-Delete of Program Dictionary record failed-see uerr_cod.
- ERR_A5A 0xA5A DTCATADI-Could not determine Dictionary file numbers
- ERR_A5B 0xA5B DTCATADI-Program Has No Relationships in ABILITY Dictionary
- ERR_5E1 0x5E1 DTPMENUS-Space Allocation Error
- ERR_5E2 0x5E2 DTPMENUS-Menu Call type must follow input criteria.
- ERR_5E3 0x5E3 DTPMENUS-Must enter Compare Criteria
- ERR_5E4 0x5E4 DTPMENUS-Must enter Call Text
- ERR_5E5 0x5E5 DTPMENUS-CURSOR=symbol..symbol not in DODA
- ERR_5E6 0x5E6 DTPMENUS-Field compare symbol not in DODA
- ERR_5E7 0x5E7 DTPMENUS-Must define USES_IMAGE first
- ERR_5F1 0x5F1 DT_MENUS-Invalid MENUS number
- ERR_5F2 0x5F2 DT_MENUS-Invalid Image number
- ERR_6C1 0x6C1 DT_REFMT-Space Allocation error
- ERR_6C2 0x6C2 DT_REFMT-Could not Open Source file

- ERR_6C3 0x6C3 DT_REFMT-Doda Length Does not match Source File
- ERR_6C4 0x6C4 DT_REFMT-Source file not a data file
- ERR_6C5 0x6C5 DT_REFMT-Source file Corrupt at open
- ERR_6C6 0x6C6 DT_REFMT-Could not read File header information
- ERR_6C7 0x6C7 DT_REFMT-Could not read Variable length record six byte header
- ERR_6C8 0x6C8 DT_REFMT-Source record READ error
- ERR_6C9 0x6C9 DT_REFMT-Write of Null Header failed after format
- ERR_6CA 0x6CA Destination found to be fixed when rebuilding variable length file.
- ERR_6CB 0c6CB Destination found to be variable when rebuilding fixed length file.
- ERR_6D1 0x6D1 DT_DFINI-Invalid Default Number
- ERR_6D2 0x6D2 DT_DFINI-Memory Allocation error on extract
- ERR_6E1 0x6E1 DT_RTREE-r-tree number not defined
- ERR_6E2 0x6E2 DT_RTREE-Associated IMAGE not found
- ERR_6E3 0x6E3 DT_RTREE-r-tree not found in extracted subset
- ERR_6E5 0x6E5 DT_RTREE-Could not allocate space for extract
- ERR_6E6 0x6E6 DT_RTREE-Could not open script work file
- ERR_6E7 0x6E7 DT_RTREE-Could not open base script file
- ERR_6F1 0x6F1 DTPRTREE-Could not allocate space for r-tree definition
- ERR_6F2 0x6F2 DTPRTREE-Could not allocate space for relations
- ERR_6F3 0x6F3 DTPRTREE-Invalid definition keyword-define USES_IMAGE(?) or define USES_SCRIPT(?)
- ERR_6F4 0x6F4 DTPRTREE-Must define USES_IMAGE and RUN_MSG USES_SCRIPT properly Check for valid image number or that you have not defined both keywords properly
- ERR_6F5 0x6F5 DTPRTREE-Invalid r-tree keyword
- ERR_6F6 0x6F6 DTPRTREE-Field symbol name not found in DODA.
- ERR_6F7 0x6F7 DTPRTREE-FIELD defined not on IMAGE defined.
- ERR_6F8 0x6F8 DTPRTREE-Could not allocate space for text
- ERR_6F9 0x6F9 DTPRTREE-One of the following keyword has a syntax error:
  USES_IMAGE(??)
  USES_SCRIPT(??)
  REPORT_PROGRAM(??)
  CALL_TYPE(??)
- ERR_6FA 0x6FA DTPRTREE-Must define substitution text

• ERR_6FB 0x6FB	DTPRTREE-Invalid Call Type
• ERR_7C1 0x7C1	DTPTABLE-Space Allocation Error
• ERR_7C2 0x7C2	DTPTABLE-Syntax Error. Disk representation and screen representation must be defined before fields
• ERR_821 0x821	DTPCALCS-Space Allocation Error
• ERR_822 0x822	DTPCALCS-Expression to long for postfix con-vertion- change DTMXWORK in DTPCALCS
• ERR_841 0x841	DTPHELPP-Space Allocation Error
• ERR_842 0x842	DTPHELPP-Syntax Error. Help text or token must be defined before fields
• ERR_843 0x843	DTPHELPP-Syntax Error. USES_SFL not define correctly
• ERR_851 0x851	DT_HELPP-Could not initlize help text subfile
• ERR_852 0x852	DT_HELPP-Subfile defined for help text not found
• ERR_7D1 0x7D1	DT_GHELP-Token used to get help text not defined in help text file
• ERR_7D2 0x7D2	DT_GHELP-File Open error. One of the needed file could not be opened. See uerr_cod.
• ERR_7E1 0x7E1	DT_BHELP-Could Not Open help text file
• ERR_7E2 0x7E2	DT_BHELP-Could Create new index file. See uerr_cod.
• ERR_1011 0x1011	DTPHOOKS-Memory Allocation Error.
• ERR_1012 0x1012	DTPHOOKS-Must define valid hook location
• ERR_1013 0x1013	DTPHOOKS-Invalid Field Name for cur_field
• ERR_1014 0x1014	DTPHOOKS-Invalid Function name
• ERR_1015 0x1015	DTPHOOKS - invalid image name foe cur_image.
• ERR_1016 0x1016	invalid keystroke for cur_keybd.
• ERR_1071 0x1071	DT_GPINN-Could not alloc memory for group record
• ERR_1072 0x1072	DT_GPINN-SIZEOF for ability type found to be zero make sure that DT_SETTY(1) was called.
• ERR_1081 0x1081	DTPGROUP-Could not allocate memory for group
• ERR_1082 0x1082	DTPGROUP-Invalid Optional Definition
• ERR_1031 0x1031	DT_FLATI-Could not open error log file
• ERR_1032 0x1032	DT_FLATI-Could not open flat file
• ERR_1034 0x1034	DT_FLATI-Could not open error log file
• ERR_1041 0x1041	DT_FLATO-Could not open Flat File
• ERR_1051 0x1051	DT_GPOUT-Could not write group to disk
• ERR_1052 0x1052	DT_GPOUT - Could not write control record.
• ERR_1053 0x1053	DT_GPOUT - Ability record has zero length.
• ERR_1054 0x1054	DT_GPOUT - Unable to delete out old definition.

- ERR_1071 0x1071 — DT_GPINN - Could not allocate memory for group record.
- ERR_1072 0x1072 — DT_GPINN - SIZEOF for ability type found to be zero make sure that DT_SETTY(1) has been called.
- ERR_1073 0x1073 — DT_GPINN - Could not read master control record.
- ERR_1074 0x1074 — DT_GPINN - No abilities found in file for this group.
- ERR_A61 0xA61 — DTCATCSP-Could not open work file
- ERR_1061 0xA1061 — DTCATDIF-Could not open work file

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V